p. 100

Introduction to IND and Recursive Partitioning

WRAY BUNTINE
RESEARCH INSTITUTE FOR ADVANCED COMPUTER SCIENCE

RICH CARUANA
SAN JOSE STATE UNIVERSITY

AI RESEARCH BRANCH, MAIL STOP 269-2
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94035
(415) 604-6527

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-91-28

October, 1991

FIA-91-27

Constraint-Based Scheduling
MONTE ZWEBEN

September 1991

The GERRY scheduling system developed by NASA Ames with assistance from the Lockheed Space Operations Company, and the Lockheed Artificial Intelligence Center, uses a method called constraint-based iterative repair. Using this technique, one encodes both hard rules and preference criteria into data structures called constraints. GERRY repeatedly attempts to improve schedules by seeking repairs for violated constraints. The system provides a general scheduling framework which is being tested on two NASA applications. The larger of the two is the Space Shuttle Ground Processing problem which entails the scheduling of all the inspeciton, repair, and maintenance tasks required to prepare the orbiter for flight. The other application involves power allocation for the NASA Ames wind tunnels. Here the system will be used to schedule wind tunnel tests with the goal of minimizing power costs. In this paper, we describe the GERRY system and its application to the Space Shuttle problem. We also speculate as to how the system would be used for manufacturing, transportation, and military problems.

FIA-91-28

Introduction to IND and Recursive Partitioning
WRAY BUNTINE AND RICH CARUANA

October 1991

This manual describes the IND package for learning tree classifiers from data. The package is an integrated C and C shell re-implementation of tree learning routines such as CART, C4, and various MDL and Bayesian variations. The package includes routines for experiment control, interactive operation, and analysis of tree building. The manual introduces the system and its many options, gives a basic review of tree learning, contains a guide to the literature and a glossary, lists the manual pages for the routines, and instructions on installation.

FIA-91-29

Acquistion and Improvement of Human Motor Skills: Learning Through Observation and Practice
WAYNE IBA

November 1991

Skilled movement is an integral part of the human existence. A better understanding of motor skills and their development is a prerequisite to the construction of truly flexible intelligent agents. We present MÆANDER, a computational model of human motor behavior, that uniformly addresses both the acquisition of skills through observation and the improvement of skills through practice. MÆANDER consists of a sensory-effector interface, a memory of movements, and a set of performance and learning mechanisms that let it recognize and generate motor skills. The system initially acquires such skills by observing movements performed by another agent and constructing a concept hierarchy. Given a stored motor skill in memory, MÆANDER will cause an effector to behave appropriately. All learning involves changing the hierarchical memory of skill concepts to more closely correspond to either observed experience or to desired behaviors. We evaluate MÆANDER empirically with respect to how well it acquires and improves both artificial movement types and handwritten script letters from the alphabet. We also evaluate MÆANDER as a psychological model by comparing its behavior to robust phenomena in humans and by considering the richness of the predictions it makes.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE Dates attached	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Titles/Authors - Attached	5. FUNDING NUMBERS
6. AUTHOR(S)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Code FIA - Artificial Intelligence Research Branch Information Sciences Division	8. PERFORMING ORGANIZATION REPORT NUMBER Attached
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Nasa/Ames Research Center Moffett Field, CA. 94035-1000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Available for Public Distribution Peter Friedland 5/14/92 BRANCH CHIEF	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstracts ATTACHED

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Introduction to IND and Recursive Partitioning

Wray Buntine, RIACS
Rich Caruana, SJSU/NASA

NASA Ames Research Center
Mail Stop 269-2
Moffet Field, CA 94035

Version 1.0
September 23, 1991

Abstract

This manual describes the IND package for learning tree classifiers from data. The package is an integrated C and C shell re-implementation of tree learning routines such as CART, C4, and various MDL and Bayesian variations. The package includes routines for experiment control, interactive operation, and analysis of tree building. The manual introduces the system and its many options, gives a basic review of tree learning, contains a guide to the literature and a glossary, lists the manual pages for the routines, and instructions on installation.

Contents

Preface

IND is a collection of C programs and C shell scripts for generating, testing, and using decision trees, class probability trees, and Bayes classifiers. IND is research software and is currently under development. First time users or those interested in obtaining the package should see the companion document "About the IND Tree Package" by Wray Buntine. Further copies of this manual or other related technical reports can be obtained by contacting:

email:	ind@kronos.arc.nasa.gov
post:	IND Tree Package C/O Wray Buntine, RIACS and Code FIA Mail Stop 269-2 NASA Ames Research Center Moffett Field, CA, 94035

The package comes with NO WARRANTY of any kind, and may not be distributed to any other party. The copyright for the package is the standard RIACS software copyright and is described in Appendix B.

IND was built on top of an early suite of software developed at Basser Department of Computer Science at Sydney University by a lineage of students of Jason Catlett: David Harper, Murray Dean, David Muller and Chris Carter, and others. Some early "man" entries and bits and pieces of the code where also done by Chris Carter while at the University of Technology, Sydney. The only program or file that remains largely intact is IND/Util/sample. All others have been recoded and restructured to a large degree, except for the "symbol" structure in SYM.h and its associated routines. We are particularly indebted to Jason Catlett's students for creating a foundation upon which to build, and to Ross Quinlan for providing the environment and ideas on which the package is based.

IND was developed by Wray Buntine, while at S.O.C.S., University of Technology Sydney ('88-'89), Turing Institute ('89) and Strathclyde University in Glasgow ('89), and code FIA, NASA Ames Research Center and RIACS ('90). More recently ('91), Rich Caruana helped make the package more presentable while on a summer internship at NASA-Ames Research Center on leave from Carnegie Mellon University.

Naturally, any deficiencies in the current software will become our responsibility, not the earlier contributors. Thanks to the various organizations above for supporting Wray Buntine's research and to the San Jose State University/NASA-Ames Research Center Research and Development Program for supporting Rich Caruana's internship. Thanks to RIACS and NASA Ames Research Center for making the package available.

PREFACE

Chapter 1

Getting Started

1.1 About This Manual

This manual is an introduction and reference manual for IND. IND is a collection of programs for generating, testing, and using classification trees and Bayes classifiers. For those not familiar with the tree literature, this manual contains an introduction to tree learning methods, a glossary, and a comprehensive bibliography. For those wishing to install the system, details are given in the last chapter, and copyright details in Appendix B.

Chapter 1 introduces first-time users to the IND software and documentation. The introduction includes an overview of this manual, a casual introduction to decision trees, a survey of the IND software, and a few sample sessions where decision trees for the hypothyroid data set are built and tested using IND. By the way, trees come in two forms, *decision trees* and *class probability trees*, where the latter replace "decisions" at the leaves with "class probabilities".

Chapter 2 surveys the more important IND runtime options. It demonstrates many of the standard IND option sets typically used when generating certain styles of trees (e.g. Bayes Trees, ID3 Trees, CART Trees). For those who are new to trees, this chapter provides useful suggestions about different ways of generating and testing trees using IND. For those who are already familiar with tree analysis, this section will quickly familiarize you with IND's options and show you how to make IND do the standard tricks. This section also provides advice on how to choose between the different methods available for the problem you are considering.

Chapter 3 is a technical introduction to decision tree methods. It is an expanded version of "An Introduction to Recursive Partitioning" written by Wray Buntine while at the Turing Institute. Those who are less experienced will find that this section provides a concise summary of decision tree methods and introduces much of the notation and many of the basic concepts required for informed use of IND. Subsequent sections of this document, as well as the man pages for IND, assume that the user is familiar with some of the methods discussed in the section. The experienced decision tree builder may wish to browse through the literature guide at the end. This has been fairly hastely thrown together so new entries are always welcome.

Chapter 4 is a copy of the man pages for IND routines. Note that it is not essential that the user be familiar with all routines for which there are man pages. The beginning of Chapter 4 suggests which man pages are likely to be useful to the typical user. Some other man pages are included for completeness, but probably will only be needed by users who intend to modify the software to make it do new tricks.

Chapter 5 contains instructions for installing IND on your machine. If IND has already been installed for you by someone else, and if you do not intend to modify IND to make it do new tricks (or fix an old trick), you can safely ignore this section. Otherwise, we suggest installing IND before reading beyond section 1.4, so that you can work through the example. This chapter also contains a brief description of where different things are located in the IND subdirectories.

Appendix A is a glossary of the terms used throughout this documentation and in the IND man pages. It may also be useful to those who have read Chapter 3, or are familiar with other tree literature, but need to reference the meaning of some terms.

The bibliography included in this manual is fairly extensive. Since IND is a research software package (as opposed to a commercial software package) and is not tutorial, some users may have to consult some of these references in order to fully understand some of the methods available within IND and their motivations or limitations. For instance, some of IND's innovative features, the Bayesian and MDL components, are based on work described in [7, 5].

1.2 Decision Trees

Decision trees are classifiers that represent their classification knowledge in "tree" form. Each interior node of a decision tree is a test on an attribute. Satisfying that test causes the instance being classified to take one branch out of that node, failing the test causes the instance to take the other branch. A decision tree is used to classify an instance by starting at the root node of the decision tree and following the path the attribute tests dictate until a leaf node is encountered. Each leaf node in a decision tree is a decision, i.e., represents a classification. An instance that ends up at some particular leaf node is classified with the class assigned to that leaf node.

For example, a decision tree for diagnosing the flu (see figure 1.1) might have leaf nodes labeled *FLU* and *NO_FLU* and might use attribute tests (on interior nodes), such as, $TEMP < 100F?$, *stomach_upset?*, and *headache?*. Each test on an attribute causes the particular instance being classified (in this case an individual with a set of symptoms) to follow one of the branches leaving that node. Eventually, since the decision tree has finite depth, the instance will end up in a leaf node labeled either *FLU* or *NO_FLU* (There may be many leaf nodes with the same label.). If the instance ends up in one of the leaf nodes labeled *FLU*, then the decision, of the decision tree for that instance, is that the individual does indeed have the flu.

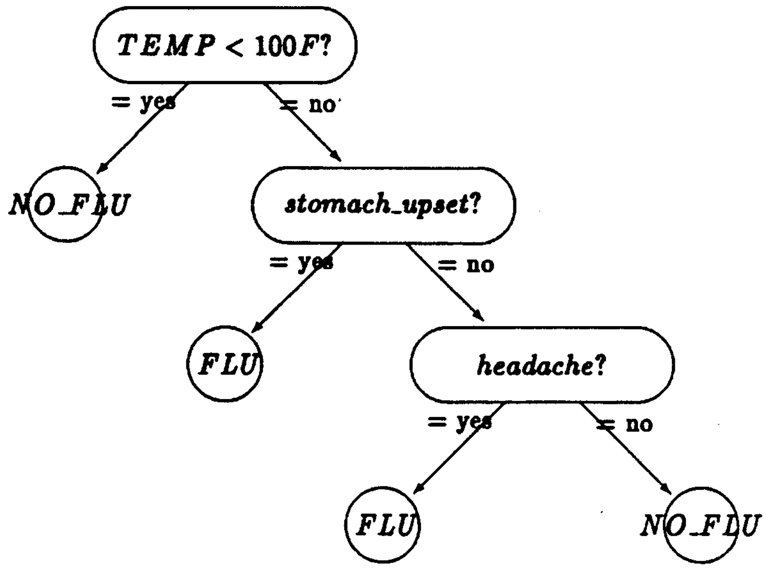


Figure 1.1: A simple decision tree for diagnosing the flu

A second kind of tree is a class probability tree. This has a vector of class probabilities at each leaf instead of a decision. For instance, the top left leaf in figure 1.1 has the decision no *NO_FLU*. This could instead be the probability vector $(0.77, 0, 23)$ (notice the elements in the vector sum to 1.0) which would represent "the probability of *NO_FLU* is 0.77, the probability of *FLU* is 0.23". This kind of tree is explained further in Section 3.3.

When we refer to "trees", we usually mean decision trees, class probability trees, or both, whichever is more appropriate.

1.3 IND - An Overview

IND is a collection of programs for generating, testing, and using trees. IND provides a potentially bewildering number of options to allow the user to precisely control how data is interpreted, how trees are grown and tested, and how results are displayed. This section is a simple overview of IND. It is intended to introduce the new user to the IND programs and to show the typical (and usually simplest) ways to run IND. More detailed information about IND can be found in the man pages for the various routines. The next chapter provides short-cuts for users who would like to use IND in specific modes.

IND consists of four basic kinds of routines: data manipulation routines, tree generation routines, tree testing routines, and tree display routines. The data manipulation routines are used to partition a single large data set into smaller training and test sets. The generation routines are used to build classifiers. The test routines are used to evaluate classifiers and to classify data using a classifier. And the display routines are used to display classifiers in various formats.

IND contains many low-level C programs that implement the basic services and a few higher-level shell scripts that encapsulate these basic services into a more user-friendly package. It is possible to use IND by directly calling the low-level manipulation, generation, test, and display programs, but this is rarely necessary; the higher-level control scripts are the correct level of abstraction for many applications of IND.

The basic control scripts in IND are **mkbld**, **mktree**, and another useful C program is tprint. There are a few even higher-level scripts that can run these basic control scripts for you, but understanding the basic scripts is important to using IND so we begin by introducing them.

mkbld is a control script that takes a data set and splits it into a training and test set. The training set is used for building the tree, and the test set is used for evaluating the performance of the tree. **mkbld** is smart enough to automatically uncompress and recompress data sets (or even build the data set using shell scripts) and allows the user to specify the sizes of the training and test set and the sampling method to use when generating them.

mktree is a control script that takes a training set and builds trees for it. **mktree** builds trees by calling an assortment of other programs that actually build the trees ("tgen"), prune them ("tprune"), and test them ("class"). **mktree** has many different options that control what methods are used to build, prune, and evaluate the trees. Tree building options include things like what the maximum tree depth should be, and what splitting rule to use (e.g., Bayes, information-gain, etc.). Tree pruning options include depth-bounded pruning with cost-complexity, pessimistic or minimum errors pruning. Tree testing options allow the user to select from several different kinds of performance measures and, for example, to control whether or not instances are classified by utility or by maximum likelihood. **mktree** passes many of the options specified to it directly to the programs it runs for you, so using the full power of **mktree** does require familiarity with the tgen, tprune, and tclass programs.

tprint takes a tree built by **mktree** and displays various kinds of information about it. For example, tprint can display the final probabilities associated with each leaf node, it can display the counts for each class at leaf nodes as well as at interior nodes, and it can even handle attributes with unknown values in any of several different ways when accumulating these counts. Of course, tprint can also pretty-print a tree in a human-readable format on a terminal.

1.4 A Session with mkbld, mktree, and tprint

In this section we demonstrate the use of IND to build and test a tree for hypothyroidism. The data set we use is the now classic hypothyroid database appearing in [33].

1.4.1 Creating the Training Set

The hypothyroid data set is contained in two files in the subdirectory IND/Data/thyroid, the *attribute_file* and the *data_file*. The attribute_file, *hypo.attr*, contains a description of the features of the data set and any special instructions on how to interpret them. For example, *hypo.attr* specifies that age is a real-valued attribute on the interval [0,100] and that sex is an ordinal-valued attribute with values M or F. The attribute file, also specifies that the classes we are trying to predict are *primary_hypothyroid*, *secondary_hypothyroid*, *compensated_hypothyroid*, and *negative* (i.e., no hypothyroidism). The format of the attribute file is described in the man page *attributes(1)*.

The file *hypo.dta* contains the hypo data set itself. Each line in this file is a single instance from the domain, coded in the language defined by the attribute file. The data set is compressed (using the UNIX "compress" command) in order to save space. Compressing data sets is optional.

Since each line in *hypo.dta* is an example from the domain, we can count the number of examples by passing an uncompressed copy of it to **wc**, the UNIX wordcount program:

```
% zcat hypo.dta | wc
  3772  113160  306174
```

From this we can tell that there are 3,772 examples in the hypo data set. The last few examples look like this:

```
% zcat hypo.dta | tail
negative 21 F f f f f t f f f f f f t f f t 0.2 t 2.5 t 108 t 1.13 t 96 f ? STMW
negative 52 M f f f f f f f f f f f f f f t 0.015 t 2.7 t 122 t 0.83 t 147 f ? other
primary_hypothyroid 78 F f f f f f f f t f f f f f f t 25 t 0.9 t 50 t 0.84 t 60 f ? SVI
negative 73 F f f f f f f f f f f f f f f t 0.95 t 2.5 t 119 t 1.04 t 114 f ? other
negative 76 M f f f f f f f f f f f f f f t 0.69 t 2.3 t 138 t 1.04 t 133 f ? SVI
negative 68 M f f f f f f f f f f f f f f t 4.8 t 2.1 t 107 t 0.99 t 108 f ? other
negative 71 M f f f f f f f f f f f f f f t 0.1 t 1.4 t 120 t 0.87 t 138 f ? other
negative 64 F t f f f f f f f f f f f f f t 0.1 f ? t 123 t 0.74 t 166 f ? other
negative 50 M f f f f f f f f f f f f f f t 0.4 t 2.8 t 94 t 0.88 t 106 f ? SVHC
negative 43 F f f f f f f f f f f f f f f t 2 t 1.8 t 121 t 0.94 t 129 f ? SVHC
```

Note that the first entry for each example is the classification for this example. Only one of these examples has primary hypothyroidism. The rest of the entries on each line are the values of the attributes in the sequence defined in the *hypo.attr* file. The fields with "?" mean the corresponding attribute value is missing. The attribute file contains the following:

```
% cat hypo.attr
class:          compensated_hypothyroid,negative,
                primary_hypothyroid,secondary_hypothyroid.
age:            cont 0..100.
sex:            F,M.
on_thyroxine:   f,t.
```

```
query_on_thyroxine:     f,t.
on_antithyroid_medication:      f,t.
sick:           f,t.
pregnant:       f,t.
thyroid_surgery:f,t.
I131_treatment: f,t.
query_hypothyroid:      f,t.
query_hyperthyroid:     f,t.
lithium:        f,t.
goitre:         f,t.
tumor:          f,t.
hypopituitary:  f,t.
psych:          f,t.
TSH_measured:   f,t.
TSH:            cont 0..600.
T3_measured:    f,t.
T3:             cont 0..100.
TT4_measured:   f,t.
TT4:            cont 0..500.
T4_measured:    f,t.
T4:             cont 0..3.
FTI_measured:   f,t.
FTI:            cont 0..400.
TBG_measured:   f,t.
TBG:            cont 0..100.
referral_source:  STMW,SVHC,SVHD,SVI,WEST,other.
```

Now we see that the first example listed above has an age of 21, is female, is not on thyroxine, etc. For more details about the format of the attribute file, see the man page *attributes(1)*.

Now that we are casually familiar with the format of the attribute file and the data file, let's build a tree. First, let's split the data set into a training set and a test set using **mkbld**. The training set will be used to build the tree, and the test set will be used to test it. Let's use 2000 examples for the training set and let the rest fall in the test set:

```
% mkbld hypo 2000
678757306

% wc hypo.bld
    2000   60000  162371 hypo.bld

% wc hypo.tst
    1772   53160  143803 hypo.tst
```

As we can see, 2000 examples went into the file *hypo.bld*, and the remaining 1772 ended up in *hypo.tst*. Note that **mkbld** *sampled* the original data base of examples, it did not just copy the first 2000 examples to one file and the last 1772 to the other.

The number **mkbld** printed out was the random seed it used for the random number generator that controlled the sampling. If we rerun **mkbld** and give it this seed (as an optional argument), it will perform an identical partitioning. This allows us to exactly replicate an experiment. If you ran **mkbld** as instructed above (i.e., without specifying a seed), then **mkbld** probably returned a different seed than the one shown above and the sampling will be somewhat different. If you wish to exactly replicate the example described here, rerun **mkbld** specifying the seed used above:

```
% mkbld hypo 2000 678757306
678757306
```

1.4.2 Building a Tree

In the last section we created a training file (the ".bld" file) and a test file (the ".tst"). Now let's build a tree using **mktree**. For now, we'll ignore **mktree**'s ability to automatically prune the tree it generates and its ability to automatically run the test set through the tree. That is we will just tell **mktree** to build the tree. Let's build the tree using the GINI index of diversity as the splitting criterion (it's ok if you don't know what that is, chapter 3 explains it) and limit the depth of the induced tree to four.

```
% mktree -o "-d4 -g" hypo
```

The "-o" argument to **mktree** says that the arguments that follow it in quotes should be passed directly to **tgen**, the program that actually generates the tree. The final argument to **mktree**, *hypo*, is the stem name. **mktree** assumes that files named *stem.attr* and *stem.bld* exist and creates a file called *stem.tree* that contains the induced tree.

Note that **mktree** did not print anything. The results of its labor are in the ".treec" file. To see the tree use **tprint**:

```
% tprint hypo.attr hypo.treec
TSH < 6.05:
|   TT4 < 48.5:
|   |   TSH < 3.5:  negative
|   |   TSH >= 3.5:  secondary_hypothyroid
|   TT4 >= 48.5:  negative
TSH >= 6.05:
|   FTI < 63:
|   |   age < 17.5:  negative
|   |   age >= 17.5:
|   |   |   age < 84.5:
|   |   |   |   T3 < 2.65:  primary_hypothyroid
|   |   |   |   T3 >= 2.65:  negative
|   |   |   age >= 84.5:  negative
|   FTI >= 63:
|   |   on_thyroxine = f:
|   |   |   T4 < 1.675:
|   |   |   |   FTI < 183:  compensated_hypothyroid
|   |   |   |   FTI >= 183:  negative
|   |   |   T4 >= 1.675:  negative
|   |   on_thyroxine = t:  negative
```

As commanded, **mktree** has limited the depth of the tree to four. The root tests to see if TSH is less than 6.05 or not. If it is, the next test is to see if $TT4$ is less than 48.5. If this is so, then the next test is to see of TSH is below 3.5. If all three of these tests are true for an example, then that example will be classified as *negative*. Other tests used in the tree are on age, FTI, $T4$, and whether or not the patient was using thyroxine.

So now let's prune this tree using pessimistic pruning and, after the pruning is complete, convert the class counts at each leaf to probabilities (Converting leaf counts to probabilities is needed for many subsequent stages of processing, e.g., for tree smoothing and even to compute the statistical summaries of tree performance. Currently, **tprune** is where this conversion is done, so you will usually want to "prune" a tree after growing it, even if you do not specify pruning options that actually reduce the size of the tree.). Rather than using **tprune** to prune the tree, let's rerun **mktree** and specify the pruning options that will be passed to **tprune** when **mktree** calls it for us:

```
% mktree -o "-d4 -g" -p "-b -e" hypo

% tprint hypo.attr hypo.tree
TSH < 6.05:  negative
TSH >= 6.05:
|   FTI < 63:  primary_hypothyroid
|   FTI >= 63:
|   |   on_thyroxine = f:  compensated_hypothyroid
|   |   on_thyroxine = t:  negative
```

Notice that the **tprint** command this time printed the tree in "hypo.tree" rather than in "hypo.treec". The distinction is important. The file "hypo.treec" stores the original unpruned tree and has example counts rather than probabilities stored at its leaves. Pruning converts this tree to the second tree stored in "hypo.tree" which we have printed in this case.

Pessimistic pruning may be well named: it pruned most of the tree! But we don't yet know which tree, the original unpruned tree or the new pruned tree, is likely to perform better on future examples drawn from this domain. So let's test the trees on the data we held aside specifically for this purpose. Again, rather than use **tclass**, the routine that actually performs the testing, let's reinstruct **mktree** to not only build and prune the tree, but to also to test it.

```
% mktree -o "-d4 -g" -p "-b" -c "-s" hypo
Percentage accuracy for tree 1 = 98.8149 +/- 0.257073
Mean square error for tree 1 = 0.0208581
Expected accuracy for tree 1 = 98.1178

% mktree -o "-d4 -g" -p "-b -e" -c "-s" hypo
Percentage accuracy for tree 1 = 98.9278 +/- 0.244665
Mean square error for tree 1 = 0.020488
Expected accuracy for tree 1 = 98.0391
```

Note that in the first run of **mktree** we specified only that the leaf nodes be converted to probabilities, but not that pessimistic pruning be done. In the second run we did specify pessimistic pruning. Looking at the performance summary for the two trees, we see that pessimistic pruning did not apparently injure the performance of the tree when tested on the 1772 examples in the test set. In fact, it may have improved the performance somewhat. This is not surprising, trees that are not pruned often *overfit* the training data.

For those of you not running these examples as you read, a Sparc 1 takes about 15 seconds to generate and test these trees, most of this time being spent in the generation stage. On some problems, the testing stage can be the most time-consuming. Obviously, regenerating the tree each time we change a pruning option is not very efficient. Why don't we first generate the tree, then prune it (sending the pruned tree to a separate file so that the unpruned tree is still there to be tested) and then, test the two trees? This is, in fact, easy to do using the routines tgen, tprune, and tclass. And if you have been reading about the options we've been using, you've already discovered that you must go to the man pages for these routines to know what options to select; the man page for mktree does not describe them. But typically, when you grow a tree you also need to prune it and evaluate it, and the ways of doing this are quite stylized. Because of this, mktree is usually the most convenient level of abstraction for using IND. It might seem a little awkward at first, but you quickly get used to. Moreover, the options used for pruning often depend on how the tree was generated, and the options used for testing the tree often depend on when it was generated and pruned, so it does make some sense to specify them all at one time.

1.4.3 A First Look at Controlling Tree Generation

Now that you are convinced that mktree is usually easier than separately running tgen, tprune, and tclass, let's do one last thing with mktree. Specifically, let's tell mktree to generate trees of depth 1, 2, and 3 and see how well they perform without subsequent pruning:

```
% mktree -o "-d0 -g" -p "-b" -c "-s" hypo
Percentage accuracy for tree 1 = 95.0339 +/- 0.51608
Mean square error for tree 1 = 0.0627282
Expected accuracy for tree 1 = 95.2003

% tprint hypo.attr hypo.tree
TSH < 6.05:  negative
TSH >= 6.05:  compensated_hypothyroid

% mktree -o "-d1 -g" -p "-b" -c "-s" hypo
Percentage accuracy for tree 1 = 97.5169 +/- 0.36966
Mean square error for tree 1 = 0.0371818
Expected accuracy for tree 1 = 96.8403

% tprint hypo.attr hypo.tree
TSH < 6.05:
|   TT4 < 48.5:  negative
|   TT4 >= 48.5:  negative
TSH >= 6.05:
|   FTI < 63:  primary_hypothyroid
|   FTI >= 63:  compensated_hypothyroid

% mktree -o "-d2 -g" -p "-b" -c "-s" hypo
Percentage accuracy for tree 1 = 98.702 +/- 0.268883
Mean square error for tree 1 = 0.022251
Expected accuracy for tree 1 = 97.9641

% tprint hypo.attr hypo.tree

TSH < 6.05:
```

```
|   TT4 < 48.5:
|   |   TSH < 3.5:  negative
|   |   TSH >= 3.5:  secondary_hypothyroid
|   TT4 >= 48.5:  negative
TSH >= 6.05:
|   FTI < 63:
|   |   age < 17.5:  negative
|   |   age >= 17.5:  primary_hypothyroid
|   FTI >= 63:
|   |   on_thyroxine = f:  compensated_hypothyroid
|   |   on_thyroxine = t:  negative
```

Obviously, we lose considerable performance by forcing the tree to be only one or two tests deep. Interestingly, the tree that is three tests deep performs worse than the smaller tree that resulted from pessimistic pruning. This is not an anomaly. Pruning a tree that has "overfit" the data often yields a better tree than simply restricting tree depth to try to prevent overfitting; pruning does not have to return trees of uniform depth and it is safer to eliminate a branch after it has proven ineffective, than it is to not expand some node before knowing if subsequent tests in the branch might make that branch useful. Of course it is not a simple matter to determine if a branch is "ineffective"; with the sometimes small amount of data appearing in the branch this is a complex statistical problem. Prior knowledge also comes into play. For instance, if you are sure that typical accuracy in prediction cannot be above 70% no matter which example is observed. and many leaves in the current branch have an accuracy of 90% (not uncommon if the tree was grown to fit the data) then it would make sense to do some heavy pruning.

Chapter 2

A Tour of IND

2.1 Introduction

Chapter 1 concluded with a session where we used IND to generate, prune, and test a few simple trees. In this chapter we flex IND's muscles a bit more. This chapter discusses IND options in detail, presents a series of IND demonstration runs, and presents a few standard sets of options that allow IND to simulate some aspects of other tree induction programs, such as ID3 or CART. This is the chapter that shows what IND can do. We begin by discussing option passing in IND. Then we discuss different kinds of IND options in more detail and, where helpful, demonstrate those options through sample runs with the hypothyroid database. We present a few different standard option sets that make IND behave similar to other tree induction programs. Finally we review some factors to consider when applying IND to a problem of your own.

2.2 Option Passing in IND

As we saw in Chapter 1, shell scripts such as **mktree** automatically run lower-level IND routines like **tgen**, **tprune**, and **tclass** for you. **mktree** passes some of the runtime arguments given to it directly to these lower-level routines. The options to be passed to these routines are specified with three flags: **-o**, **-p**, and **-c**. These flags introduce the options for **tgen**, **tprune**, and **tclass**, respectively. The options you wish to pass follow these flags, usually as a string enclosed in quotes. It is important to include "-" signs in these option strings. An example will make this clearer. Suppose we wish to run **mktree**, telling it to use **tgen** options "-g", "-U 3", and "-d 4", to use **tprune** options "-b" and "-e", and to use **tclass** options "-g", "-p", and "-s". While we're at it, let's also specify the **mktree** options "-a" and "-D". This could be specified in several different ways, two of which are:

```
mktree -a -D -o "-g -U 3 -d 4" -p "-b -e" -c "-g -p -s" hypo
mktree -a -D -o "-gU3 -d4" -p -be -c -gps hypo
```

Notice that in the second, options without arguments are strung together. The following ways, however, are incorrect:

```
mktree -aD -o "-g -U 3 -d 4" -p "-b -e" -c "-gps" hypo
mktree -a -D -o "-gU3-d4" -p "-be" -c "-gps" hypo
```

The first is incorrect because of "-aD": neither **mkbld**, **mktree** or **ttest** can string option arguments together. The second is incorrect because the option argument to "-o" has no space after the "3".

The option **-D** tells **mktree** display the commands it executes. This option is supported by most IND shell scripts and is very handy when debugging or when learning how to use IND. Running either of the correct commands invokes the following sequence of commands:

```
limit datasize 12m
limit stacksize 12m
limit cputime 2000
tgen -g -U 3 -d 4 hypo.attr hypo.bld ./hypo.treec
tprune -b -e hypo.attr ./hypo.treec
mv ./hypo.treec.p ./hypo.tree
tchar hypo.attr ./hypo.tree ./hypo.ctr
tclass -g -p -s hypo.attr ./hypo.tree hypo.tst
```

The limit commands restrict how long **tgen** can run and how much memory it can use. **tgen** will quit gracefully when any of these limits are exceeded. Note that if you run either form of the **mktree** command above, you will see a lot of shell execution detail that has been deleted from the execution sequence printed above. This "sanitization" was done only to make it easier to see what IND commands **mktree** is executing.

2.3 Building Trees

IND is capable of building and using several different kinds of trees. The basic tree is a conventional decision tree using perhaps the GINI index of diversity as its splitting rule (i.e. , it uses the GINI to determine which test to install when expanding a node). But IND can also generate other kinds of trees. For example, IND can use a Bayes splitting rule instead of the GINI index of diversity and, thereby, build Bayes trees. IND can generate option trees which are a representation of many alternative trees in an and-or structure, see Section 2.5.5. This section briefly examines the various types of trees available in IND.

2.3.1 Splitting Criteria

IND can use several different criteria when evaluating the quality of different tests. The available options include the GINI index of diversity ("-g"), the Bayes splitting rule ("-t"), and information gain (the default).

Lookahead during splitting is invoked with the "-B" option to **tgen**. This starts a depth-bounded beam search to look for the best node. You should only do this with the Bayes splitting rule. For instance, "-tB3,5,0.00001" does 3-ply lookahead with a beam width of 5, and at each search point only expanding nodes within a factor 0.00001 of the best.

IND allows multi-valued attributes to be binary encoded with the "-S2" option in **tgen**. This means if a multi-valued attribute A has 6 values, 0,1,2,3,4,5, then instead of producing a test A? on the attribute with 6 outcomes, depending on the value of A, allow one of 6 tests of the form $A = 3$?, with outcomes *true* and *false*.

IND also allows subsetting of multi-valued attributes with the "-S0" option in **tgen**. This means if a multi-valued attribute A has 6 values, 0,1,2,3,4,5, then instead of producing a test A? on the attribute with 6 outcomes, depending on the value of A, allow one of many tests of the form $A \in \{1,3,5\}$ or $A \in \{0,5\}$ with outcomes *true* and *false*.

2.3.2 Stopping Rule Options

tgen and, therefore **mktree**, can be told to limit tree depth to a certain size. This is done with **tgen**'s "-d depth" option. Note that a depth of zero means that the tree has only one attribute test (at the root node), a depth of one means that there is a subsequent level of attribute tests just below the root node test, etc. See section 1.4 for an example of building trees with different depth bounds.

Another stopping rule is **tgen**'s "-s min" option. A node with fewer examples is automatically made a leaf.

Various "pre-pruning" stopping rules can also be programmed using **tgen**'s "-J" option with the *leaf-fact* factor, when growing Bayes trees. To do pre-pruning of Bayes trees use:

```
mktree -o "-tJ1,0.005,0.75,1.0"  -p -b  -c -slvg  hypo
```

The factors "0.005,0.75" are search parameters and not important in this case. The 1 in "-J1" says we are doing Bayes trees and not option trees; option trees have multiple (optional) tests. The pre-pruning factor here is the 1.0. Use a factor of 1.0 to stop when the best test is as good as the leaf, a smaller factor is more cautious in stopping, a larger factor pre-prunes more severely.

2.3.3 Pruning Options

Cost-complexity pruning, for various reasons, is done in **tgen**, see Section 2.5.1, but can also be done with the "-c" and "-V" options in **tprune**. Pessimistic pruning is done with the "-e" option to **tprune** and minimum errors pruning with the "-M" option.

Other tree operations which perform the same service as pruning are smoothing, the "-b" option to **tprune**, which averages over pruned subtrees, and choosing the maximum a posterior tree using the "-B" option to **tprune**. In general the "-b" option should give better class probability estimates but can be slower. These methods should be used along with careful use of the prior options. See Section 2.5.4, 2.5.3 and 2.6.1.

2.3.4 Prior Options

IND can be primed to handle three different kinds of prior knowledge when growing trees.

Structural constraints: The contexts feature (see the man entry *attributes(1)*) allows structural constraints to be specified on the forms of trees that can be built.

Preference for simpler trees: Typically you might have expectations such as: there are many irrelevant attributes; all attributes give some guide as to the class; a small classifier should perform quite well, etc. In these cases you should set the "-P" option carefully.

Typical prediction accuracies: What sort of predictions do you expect to make about class? In some cases you know accurate prediction is very difficult, in other's accurate prediction should be quite feasible. The "-A" option should be used here.

Without use of the "-P" option, you are saying *aprior* that all trees are equally likely. This means you do not believe there are many irrelevant attributes, and in fact you believe most attributes contribute somehow to the class. Using "-P-0.693,-0.693" for a binary tree corresponds to saying that adding a new test instead of a leaf makes the resultant tree 4 times less likely. Using "-P0,-0.693" for a binary tree corresponds to saying that a tree with 6 leaves is as twice as likely *aprior* as a tree with 7 leaves, and $8 = 2^3$ times more likely than a tree with 9 leaves. Using "-P-0.693,-0.693,02" corresponds to an even more extreme preference for smaller trees, as typically done when "encoding" a tree.

The setting of the *alpha* parameter to the tree priors is done with the option "-A *alpha*". We set this parameter from our *aprior* expectations about class probabilities at leaf nodes. If you expect leaf nodes to be highly accurate in their predictions, then you should use "-A1". This means you expect to see class probabilities at leaf nodes to be extreme (i.e., one class will have a probability near one and all the others will be low). If you have no expectation that probabilities will be either low or high, then you have uniform prior. In this case set "-A2" in the two-class case, "-A4" in the four-class case, etc. If you expect prediction accuracies to be poor, and to be little changed from the base rate class probabilities, then use an even higher value of alpha, such as "-A6" in the two-class case, or "-A12" in the four-class case, etc.

2.3.5 User Override Mode

A powerful option provided by **tgen** is "-o", the *manual control* option. This option allows the user to manually overide all decisions as each node is expanded and to examine some of the statistical information **tgen** has computed for each possible test. This allows you to figure out *why* **tgen** decided to install some particular test at some particular node. More importantly, it also allows you to *control* what test to install at particular nodes. In this way you can manually build a tree using **tgen** to supply you with relevant statistical information, but reserving the actual decision making for yourself. The result will be a tree in the syntax appropriate for IND that you have been able to exert control over. It is possible to use manual control to build entire tree, but this could be tedious. Fortunately, you have the option of allowing **tgen** to install the test it would pick at any node, and even to have **tgen** complete growing specific subtrees by itself. A sample execution trace using manual control is shown below.

```
% tgen -ot hypo.attr hypo.bld hypo.tree
Interact? (type 'h' for help): h
Interaction:
 'n' = no, continue growing,
 's' = no interaction for subtree, 'w' = none for parent tree.
Modify growing:
 'a' = abort growing and save tree so far,
 'c' = choose a test at this node, 'f' = force leaf.
Reports on this node:
 'l' = list tests at node, 'o' = give report on options,
 'g' = print gains, 'e' = print error est.
 'r' = give full report on current stored tests,
 'x' = toggle on/off plotting of attribute gains,
 'k' = kill attribute gain graphs.
Reports on tree:
 'q' = print subtree so far, 'p' = print tree so far,
 'u' = print statistics on tree so far,
Interact? (type 'h' for help):
```

2.4 Miscellaneous Options

2.4.1 Data Handling Options

Most of IND's data handling options apply to **mkbld** and are fairly well explained in the **mkbld** and **sample** man pages. We'll mention a few of them here just so you know what is available.

By default, **mkbld** does sampling without replacement (unless the data set is in a ".all" file; see the man page *attributes(1)*). With the "-r" option you can tell it to sample with replacement.

With the option "-p m i", **mkbld** can be told to partition the data set (in the ".dta" file) into m partitions and to use the i-th such partition as a test set (the ".tst" file) and the remaining m-1 partitions as the training set (the ".bld" file). With the option "-P m i", **mkbld** will partition the ".bld" file itself as described. The "-p" option is useful when training on the entire sample and you wish to use a (somewhat slow implementation of) cross-validation to estimate the error. The second in useful when doing cross-validation error estimation in comparative studies where one often looks at performance on subsamples.

With the "-c proportion" option, **tgen** can be told to use *proportion* of the examples as the training set and the remaining as the test set for cost-complexity pruning. The default proportion is 0.7.

2.4.2 Tree Evaluation Options

IND provides a number of performance measures and summary statistics to aid in evaluating a tree.

If you would like to evaluate how a tree performs on a particular test set, then use the "-t" option in **tprint**, as well as other options you might like. For instance use:

```
tprint -bp -t hypo.tst hypo
```

This will print out the class probabilities used by the classifier **tclass** at each leaf, together with a breakdown of how the test set "hypo.tst" faired on each leaf in the tree.

If you have grown several different trees using different splitting rules, and you would like to estimate which would be the better tree, then you can use the log. posterior measure printed out using "**thead** -s hypo.tree" or "**tclass** -g ...". This is particularly useful when working with Bayes trees or option trees which have been grown specifically to maximise this measure. If memory or CPU limits were overran during tree growing, then it is important to check the log. posterior to ensure the tree grown is reasonable by comparison.

Finally trees grown by a specific method (e.g., CART-like or C4-like) can be compared by using the cross-validation or repeated resampling features of **ttest**.

2.4.3 Classifying a New Example

Given a new unclassified example, or a set of the same, you may wish to use IND to predict the class or estimate a class probability vector for the new example. Do this as follows: place the example as a single line in a file as you would for the data used to grow a tree. Suppose this is in "hypo.new".

```
% cat hypo.new
negative 78 F f f f f f f f t f f f f f f t 25 t 0.9 t 50 t 0.84 t 60 f ? SVI
```

Notice that the new example has been assigned the (arbitrary) legal class value *negative*, however, this is just to prevent **tclass** from complaining, and will be ignored. Now run

```
% tclass -dp hypo.attr hypo.tree hypo.new
primary_hypothyroid      0.0200611+0.0423564+0.917523+0.0200598
```

The first field printed is the predicted class. The second set of fields is the class probability vector. Notice the third probability is the largest so **tclass** has predicted the third class listed in the file "hypo.attr", which is *primary_hypothyroid*. If in addition, the "-v" option is used with **tclass**, then posterior variances are printed as well. These estimates are usually improved with the "-b" option to **tprune** and with careful choice of the prior options when using Bayes trees or option trees.

2.4.4 Tree Display Options

Various details of a tree can be printed out. The most important detail to print is a basic decision tree. Suppose we have just built a tree using the Bayes splitting rule and Bayesian smoothing:

```
mktree  -o -t  -p -b  hypo
```

To find out what the basic decision tree generated looks like we can run either of the commands

```
tprint hypo
tprint hypo.attr hypo.tree
```

This takes the class probability tree stored in the binary file "hypo.tree" and displays the tree as was done in Section 1.4. The single form requires only specification of the stem. The double form allows you to display a tree stored in some other file, such as a counts tree "hypo.treec".

The class probability tree would be printed using "**tprint** -p hypo". If the tree has been smoothed using the "-b" option in **tprune** then you should really do "**tprint** -bp hypo". The first form prints the class probabilities at that node. The second form prints the probabilities that would be used during classification (these are different only for a smoothed tree, i.e. one pruned using **tprune** -b), but is implemented in a rather slow manner. Print them both out and compare on a few simple trees. Use "**tprint** -ipq hypo" to find out how the calculations differ. This would yield a tree like:

```
% mktree -o -td3 -p -b hypo
% tprint -ipq hypo
 0.04092 0.9336 0.02445 0.000998 negative (L0)
TSH < 6.05:  0.0006109 0.9976 0.0006109 0.001222 (L0.999999)
|   FTI < 338:  0.0006423 0.9981 0.0006423 0.0006423 (L1.4157e-06)
|   FTI >= 338:  0.0119 0.9524 0.0119 0.02381 (L6.65055e-07)
|   |   query_hypothyroid = f:  0.0125 0.9625 0.0125 0.0125 (L7.50643e-07)
|   |   query_hypothyroid = t:  0.125 0.5 0.125 0.25 (L4.41554e-07)
|   |   |   sex = F:  0.1667 0.5 0.1667 0.1667 (L3.09088e-07)
|   |   |   sex = M:  0.1667 0.3333 0.1667 0.3333 (L3.09088e-07)
TSH >= 6.05:  0.221 0.6442 0.1321 0.002695 (L0)
|   TSH_measured = f:  0.004878 0.9854 0.004878 0.004878 (L1)
|   TSH_measured = t:  0.4824 0.2235 0.2882 0.005882 (L0)
|   |   FTI < 64.5:  0.01887 0.09434 0.8679 0.01887 (L0.434369)
|   |   |   thyroid_surgery = f:  0.02041 0.06122 0.898 0.02041 (L0.565631)
|   |   |   thyroid_surgery = t:  0.125 0.375 0.375 0.125 (L0.565631)
|   |   FTI >= 64.5:  0.6777 0.281 0.03306 0.008264 (L8.46707e-18)
|   |   |   on_thyroxine = f:  0.8723 0.07447 0.04255 0.01064 (L1)
|   |   |   on_thyroxine = t:  0.03226 0.9032 0.03226 0.03226 (L1)
```

The weights printed in brackets after the "L" indicate the posterior probability that this node will be a leaf. **tprune** and **tclass** use these weights to compute the weighted average of the class probability vectors along a branch. For instance, examples falling down the branch with test outcome $TSH < 6.05$ all use the probability vector for that top node, (0.0006109 0.9976 0.0006109 0.001222), because the weight for that node is 0.999999.

Class counts themselves can be printed using "**tprint** -c hypo.attr hypo.treec". Notice the class counts tree "hypo.treec" needs to be used, not the class probability tree "hypo.tree". If you find the tree is way too large, then you might like to print it out only to a fixed depth, such as depth 2 using "**tprint** -ciD2 hypo". Option "-i" in this case ensures internal nodes will have details printed as well as the leaf nodes.

2.4.5 Tree Editing

You can always edit a grown tree using **tchar**. To do this, first convert the tree to character format using **tchar**. Then delete/change nodes as you see fit. The character format of the tree is explained in the manual entry *tchar(1)* and you can see it yourself by printing the same tree using **tprint** and comparing. Finally, convert the tree back using **tchar -a**. If you are working with an unpruned (counts) tree, then it is safer to use "**tchar** -ac" as this will check and correct all the intermediate node counts for you.

2.4.6 Handling Attributes with Unknown Values

Trees have trouble with attributes that sometimes have unknown values: which branch do we send the example down if the example does not have a value specified for the attribute to be tested at the test node? Yet in some domains (e.g. , Medical Diagnosis) it is completely impractical to require that each patient have every test performed. IND does not implement the "surrogate test" feature of CART to handle missing values, however it does have a number of different ways. The default is a method that performs fairly well in general if you do not wish to be concerned with the other variations.

IND can handle examples with unknown attribute values in a number of ways. By default (**tgen** -U1) IND sends the example down each branch with the proportion found in the training set at that node. In effect, IND *splits* the example into fractional examples, with the larger piece going down the branch most of the data follows. (This does not pose any conceptual problems because all that is really needed at leaf nodes is the count for each class, and it doesn't matter very much if that count contains fractional examples.)

Instead of the default, IND can handle unknown attributes by sending the example down the branch of the tree most commonly taken by other examples. In effect, IND is assuming the missing attribute value is the same as the most common value seen for that attribute, at that node, in the training data. IND can also be told to send examples with unknown attribute values down the branch chosen with probability proportional to that found in the training set at that node. That is, if 80 percent of the examples at this node with known attribute values take the left branch, IND will send an example with an unknown attribute value down the left branch 80% of the time. Alternatively, IND can send the entire example down the branch that most of the examples went down, or it can send the entire example down a single branch picked with probability equal to that of the proportion of examples that went down that branch.

IND's way of dealing with unknowns is uniform between the different routines, i.e. , the same options are available in **tgen**, **tprune**, and **tclass** and they are all specified the same way. See the man page for **tgen** to see how to select from the different options. If you are using **mktree**, be sure to specify the same option for generating, pruning, and testing unless you really want to handle unknowns differently in the different phases.

2.4.7 Echoing Shell Scripts

Many IND shell scripts accept option **-D**. This causes them to echo the shell commands they execute. This is quite useful for debugging and also for learning more about the IND routines at the lower level of abstraction.

2.4.8 Comparative Trials

IND provides a framework for doing repeatable comparative trials of learning algorithms. The **ttest** C shell script is used for controlling partitioning, tree growing and testing. This outputs statistical data (accuracies, both actual and predicted, tree size, etc.) to trial files that can be subsquently processed by a program such as **lstat** to check for statistical significances.

2.5 Standard IND Option Sets

With the appropriate choice of options, IND will simulate a variety of other tree induction methods. This section presents the options you should use to make IND behave similar to other tree induction programs such as **CART** or **ID3**. The presentation here is terse, only explains options not covered earlier or that might be nonobvious. This section is valuable, mainly because it tells you what *combination* of options to use.

2.5.1 CART Style

Standard **CART** uses the GINI index of diversity when splitting, and does cost-complexity pruning and 10-fold cross-validation.

To achieve this with IND, we select GINI and 10-fold cross-validation ("-gC 10"), and use cost-complexity pruning with the number of standard deviations set to 0.0 (the so-called 0-SE rule) or 1.0 (the so-called 1-SE rule). The "-A0.0001" option to **tgen** cause it to use probability estimates at nodes that are practically equivalent to **CART**'s simple frequency probability (computed in IND via the Laplacian formula using a value for *alpha* that is so small that it behaves effectively like 0.0 but avoids potential division by 0.0). The **tprune** option direct **tprune** to prevent subsequent Bayesian averaging in **tclass** by setting the leaf probabilities to 1. Finally, **tclass** is told to print out a summary of performance for the induced tree.

```
%  CART-like  with 0-SE rule and subsetting
mktree -o "-gC10 -p0 -A0.0001 -S0" -p -n -c -sl hypo

%  CART-like  with 1-SE rule and no subsetting
mktree -o "-gC10 -p1 -A0.0001" -p -n -c -sl hypo
```

2.5.2 C4 Style

The early version of C4 used pessimistic pruning and the information gain splitting rule. Subsetting ("-S0") or binary encoding ("-S2") could be used by **tgen** if so desired.

```
% C4-early with no subsetting
mktree -o "-u -A0.0001" -p "-en" -c "-sl" hypo
```

2.5.3 Minimum Encoding Style

With minimum encoding, we seek to grow the tree that has the "minimum encoding" of tree plus data given tree (see Section 3.4.4). The implementation in IND for these methods is not quite standard because cut-points are not encoded according to any of the standard encoding schemes for trees. There is, however, a Bayesian "discounting factor" for cut-points (given in [7]), which probably has a perfectly acceptable encoding interpretation.

The "-B" option to **tprune** ensures the single best tree will be chosen. The "-A1" option to **tgen** uses a prior on probabilities at leaf nodes that expects extreme probabilities at leaf nodes but is symmetric in the sens that no class a *a prior* better than any other. To use a uniform prior (all leaf class probabilities are equally likely), then use "-A*C*" where *C* is the number of classes. See Section 2.3.4 for other ways of setting this.

The option "-P-.693,-693,02" to **tgen** does the real work. This gives leaves and nodes alike a weight of $\log 0.5$, which means we give a single branch a probability of 0.5 of being length 1, of 0.25 of being length 2, of 0.125 of being length 3, etc. This also encodes the tests made at each node with the "02" flag at the end.

```
%   basic MDL-like
mktree -o "-utP-.693,-693,02 -A1'' -p -B -c -slvgQ hypo

%   MDL-like with normalising the tree prior
mktree -o "-utNP-.693,-693,02 -A1'' -p -B -c -slvgQ hypo

%   MDL-like with uniform class priors in 2-class case
mktree -o "-utP-.693,-693,02 -A2'' -p -B -c -slvgQ hypo

%   MDL-like with maximising alpha
mktree -o "-utP-.693,-693,02 -A2 -W3,1'' -p -B -c -slvgQ hypo

%   MDL-like with 3-ply breadth-5 lookahead
mktree -o "-utP-.693,-693,02 -A1 -B3,5'' -p -B -c -slvgQ hypo
```

2.5.4 Bayes

Bayes trees is essentially MDL-like, but with a more flexible interpretation and a more thorough theoretical basis.

Rather than trying to choose the single best tree, smooth over several trees using the "-b" option to **tprune**. Assuming that the "-A" and "-P" options are set reasonably well, this option almost always improves (on average) class probability estimates and often prediction accuracies.

Also, options to **tgen** which essentially set prior parameters ("-A" and "-P") should be chosen as your prior dictates, rather than due to some notion of "the shortest encoding". See Section 2.3.4. Set alpha as described above. This is critical and effects the performance of the resultant tree considerably. For instance, if expected errors printed by **tclass** are significantly higher than actual error on the test set, then you probably have alpha set to low. If you believe there is considerable structure in the problem, and that several of the attributes are important when predicting class, then you probably should not be using the "-P" option because this is quite an extreme *aprior* preference for shorter trees. The classic LED problem is a case in point where all LEDs are moderately indicative of the digit so short trees should not be expected *aprior*.

```
%    basic Bayes trees
mktree -o "-utA1" -p -b -c -slvg hypo
```

Many variations exists, as for MDL-like. You can include subsetting, lookahead, etc.

2.5.5 Option Trees

Option trees extends Bayes trees by growing many different trees and storing them in a compact and-or graph structure. It tends to be time consuming and memory consuming, although the improvement in prediction accuracy can be quite significant [5]. Advice is given in the man entry *tgen(1)* and *mktree(1)* on how to control this. Hopefully, this will be cleaned up in later releases of IND. In general, use of option trees, where computationally feasible and with an appropriate choice of prior parameters, should yield the best prediction accuracies and class probability estimates for all the tree methods in IND. On very large problems, option trees are not currently practical.

To use option trees, first try Bayes trees with careful choice of the prior parameters to get a reference point. Make a note of the log. posterior of the tree grown (see Section 2.4.2) as the option tree will only be any good if it gets a higher log. posterior. Now add the "-B", "-J" and "-K" options as explained above. Experiment with different depth bounds and factors to prevent memory or CPU overruns, as explained in the man entry *tgen(1)*. If the option tree grown has a log. posterior no better than for the Bayes tree, then the search is causing problems. For very large data sets (such as the "nettalk" data set) it is not currently realistic to grow option trees.

2.6 Choosing Options

In general, good performance from a tree package with as many options as IND requires careful choice of the right set of options. This section reviews some basic features and procedures you should go through when applying IND to a new problem.

One thing this section does not do is explain the general steps you have to go through when applying a supervised learning system such as IND: steps like gathering the right data, choosing a set of attributes, enlisting the help of a domain expert, etc. For this kind of general introduction, see, for instance [33, 18, 22, 10].

2.6.1 Prior knowledge

In most applications, you have some vague prior knowledge that could be of use when building a tree. IND can be primed to handle three different kinds of prior knowledge as described in Section 2.3.4. If you have moderate confidence in your setting of these, then Bayes trees or options trees are the best algorithms for you to try. Try Bayes trees first then experiment with option trees to see if you can grow a tree with improved log. posterior probability.

2.6.2 Benchmark methods

Both CART and C4 are widely regarded as being good tree algorithms. If your disposition is such that you would rather use an "accepted" or "standard" benchmark method, then you should choose the appropriate option set to mimic these.

2.6.3 Appropriateness of Trees

Before choosing a tree method at all, you should consider whether such a method is indeed appropriate. Trees do not represent DNF rules or linear classifiers (for instance, logistic regression or a perceptron) very well. In general, you may wish to try several different supervised learning methods on your problem and compare them. If classification seems to require weighing up many different factors, then trees will probably be poor classifiers.

2.6.4 Parsimony or Understandibility

In same cases it is important that the tree built be presentable to a human audience. This means a far shorter tree is superior even if it has a slightly less prediction accuracy.

To achieve parsimony with CART-like options, use the 1-SE rule instead of the 0-SE rule. To acheive parsimony with trees smoothed using "**tprune -b**", such as when using the MDL options, Bayes trees or option trees, use the "-P' option to ensure a greater preference for smaller trees. This means making either the node or leaf weights more negative, or using the 02 flag instead of no prior flag at all.

Chapter 3

Learning Tree Classifiers

3.1 Introduction

This chapter provides a detailed, somewhat mathematical treatment of, methods for learning tree classifiers or "decision tree induction", as it is popularly called. It is not necessary to fully understand the mathematics in this chapter, but an intuitive understanding of the issues and familiarity with some of the terminology is required to use IND well. The Bayesian tree learning techniques that make up much of IND's new features are described elsewhere [7, 5].

Section 3.2 is a brief introduction to the problem of classifier induction. This section does not survey the area nor discuss the basic issues. Instead, it serves mainly to set the stage for the learning problem that will be addressed in the subsequent sections.

Section 3.3 reviews the "standard" methods for learning trees. This includes methods such as Quinlan's ID3 [29], C4 [33, 31] and CART [3]. This section ends with a sample tree induced for the noisy LED problem.

Section 3.4 is a brief guide to research in the area of learning trees. The Bayesian tree learning techniques that make up much of IND's new features are described elsewhere [7, 5].

3.2 Introduction to Learning Classifiers

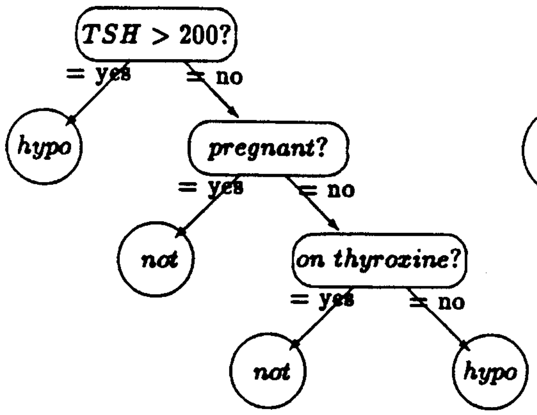
The learning of classification rules from data is performed as an aid to knowledge acquisition [33] We typically have an *expert* who is sufficiently knowledgeable to formulate the problem for us and is in possession of a *training set*, a set of examples each belonging to one of a small discrete set of mutually exclusive and exhaustive *classes*. Classes might be "positive" and "negative", or "diseased", "healthy" and "recovering", or similar. The task is to develop a *classification rule* to predict the *class* of further, unclassified examples.

The problem formulation is as follows: The *examples* are grouped into different *types*. In a given problem a particular type of example is usually associated with a particular *description* in terms of an expert-supplied *language*, consisting of 10–30 *attributes*. Each attribute may be binary ("true" or "false"), multi-valued or real valued [29].

Quinlan *et al.* [33] present an induction problem where examples correspond to patients that attended a laboratory for endocrine analysis. Each patient is described in terms of attributes such as sex, age, pregnant and on-lithium. Two patients are considered to be of the same type if they have the same attribute values. One binary classification of patients is whether they are "hypothyroid" or "not hypothyroid", and the corresponding task is to predict this given other details of the patient. The training set available for this problem is a set of some 4000 recent medical records.

3.3 Trees

Methods for learning decision trees and class probability trees are found in both machine learning and applied statistics, and have been under development in some form or another for some two decades. This chapter reviews a cross section of current methods, develops alternative Bayesian approaches, and makes a comparison of the two families. One standard technique for building classification rules from data is the so called recursive partitioning algorithm that forms the basis of systems such as ID3 [29] and CART [3]. These algorithms build a decision tree such as the one shown in the left side of Figure 3.1. The tree shown on the left has the classes *hypo* (hypothyroid)



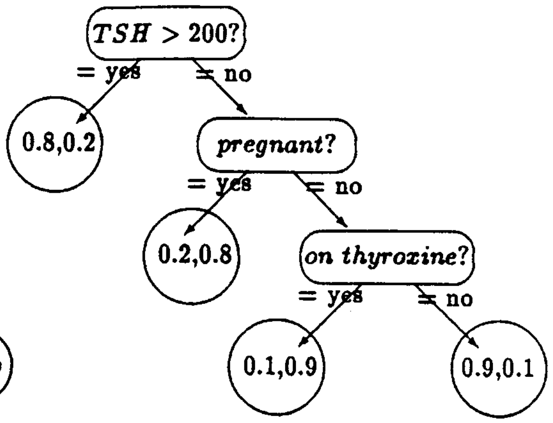


Figure 3.1: A decision tree and a class probability tree from the thyroid application

and *not* (not hypothyroid) at the leaves. This tree is referred to as a *decision tree* because decisions about class membership are represented at the leaf nodes. This is the kind of tree you see when you run **tprint** without any options. Notice that the real valued attributes TSH and FTI have been incorporated into the tree by making a binary test of the form *attribute* < *cut-point*. Also, the tree need not be binary; if an n-valued attribute is tested at one of the nodes, then the tree might have n branches coming from the node, one for each value.

In typical problems involving noise, class probabilities are usually given at the leaf nodes instead of class decisions, forming a *class probability tree* (where each leaf node has a vector of class probabilities). A corresponding class probability tree is given in the right of Figure 3.1. The leaf nodes give predicted probabilities for the two classes. This is the kind of tree you see when you run **tprint** with the "-p" or "-bp" options. Notice that this tree is a representation for a conditional probability distribution of class given information higher in the tree. This statistical interpretation of a tree is used as the basis for a statistical analysis of tree learning [5].

3.3.1 Recursive partitioning

The basic algorithm builds a tree top down using the standard *greedy search* principle; always take the perceived best move and do not bother searching to find a better one. This results in an algorithm whose running time is typically linear or log-linear in the number of examples. That is, given a sample of N examples, running time will be $O(N)$ or $O(N \log N)$ respectively.

As each node is being built the subset of training examples that would belong at that node is considered. The basic algorithm can be summarized as follows:

1. Find out how many of the training examples belong in each class. We shall refer to this information as the node statistics.
2. If all training examples belong to a single class, or if some other *stopping rule* applies, the tree is a leaf labeled with that class.
3. Otherwise,
   (a) select a test using a *splitting rule*, based on one attribute, with mutually exclusive outcomes;

(b) *divide* the training set into subsets, each corresponding to one outcome, and

(c) apply the same procedure to each subset

Sometimes, the resultant tree is modified at the end, for instance by *pruning* back branches into leaves [31]. This means there are three important sub-routines for the recursive partitioning algorithm:

Stopping rule: should the current node be grown or turned into a leaf. IND options are described in Section 2.3.2.

Splitting rule: which is the best test to make at the current node, IND options are described in Section 2.3.1.

Pruning rule: how should the tree be pruned after it is grown (sometimes called *post-pruning*). IND options are described in Section 2.3.3.

Once constructed, such a decision tree can be used to classify a new unclassified example described in terms of the same attributes. This is done by tracing through the tree recursively to find in which leaf the example should belong.

3.3.2 Stopping rules

Stopping rules were originally called *pre-pruning* rules when people originally tried using statistical measures to predict if further growing was unnecessary, for instance using the chi-squared statistic. These are sometimes ineffective because a tree has to be grown out before any advantage is realised. A Bayesian variation this statistical pre-pruning is described with the "-J" option in Section 2.3.2.

Most recent algorithms stop growing trees when certain fail-safe conditions are satisfied.

- The node is "pure", it only contains examples of the one class. IND always does this.
- The node is greater than a certain depth. This is the "-d" option to **tgen**.
- The node has less than 5 (say) examples. i.e., any smaller figure gives insignificant estimates of class probability. This is the "-s" option to **tgen**.

3.3.3 Splitting rules

A splitting rule typically works as a one-ply lookahead heuristic. For each possible test, build leaf nodes at each of the branches and then evaluate the test according to some heuristic such as maximum information gain [29]. This approach can be summed up as follows:

1. Based on each test being evaluated, divide the training set into subsets, each corresponding to one outcome;
2. construct a leaf node corresponding to each outcome and take the node statistics at that leaf;
3. this yields a complete subtree of depth 1 at the node being evaluated; finally
4. evaluate the quality of this subtree using some statistical heuristic such as information gain.

	class c_1	class c_2	...	class c_C	total
test outcome 1	$n_{1,1}$	$n_{1,2}$	...	$n_{1,C}$	$n_{1,\cdot}$
test outcome 2	$n_{2,1}$	$n_{2,2}$	...	$n_{2,C}$	$n_{2,\cdot}$
⋮					
test outcome T	$n_{T,1}$	$n_{T,2}$	...	$n_{T,C}$	$n_{T,\cdot}$
total	$n_{\cdot,1}$	$n_{\cdot,2}$	...	$n_{\cdot,T}$	$n_{\cdot,\cdot}$

Table 3.1: A Class×Outcome Counting Table

The test that yields the highest evaluation is chosen.

The evaluation process can also be looked at in terms of a table. At each node where a test is to be selected, the task is first to construct a table counting the number of training examples that occur in each class for each outcome of the test, and second, to calculate a statistical heuristic on the table to estimate the quality of the test. A prototypical table built is given in Table 3.1. Here $n_{i,j}$ corresponds to the number of examples at the node being evaluated that fall in test outcome i and have class j, $n_{\cdot,j}$ is the number that have class j regardless of test outcome, and $n_{i,\cdot}$ is the number that have test outcome i regardless of class. If we also have to consider the case where the outcome is unknown for some training examples (because an example does not have a particular attribute value given), then an additional entry "outcome unknown" needs to be added to the table before the total row.

For a simple test on a multi-valued attribute, "test outcome i" corresponds to *attribute-value* $= v_i$, for each distinct value v_i of the attribute. For a test constructed as a binary cut-point on a real valued attribute, the two test outcomes correspond to *attribute-value* $<$ *cut-point* and *attribute-value* $\geq$ *cut-point*. Many other test types are possible, but these two are representative and so are sufficient for our initial investigation.

The statistical tests commonly used are intended to favor splits that yield rows having significantly different class distributions. These are mostly similar to the chi-squared statistic for testing dependence in a contingency table. Some common tests are:

Information gain: maximise the information gained about the class by making the test [29]; this is the default splitting rule in IND:

$$I(class|test) = \sum_{i=1}^{T} Pr(outcome\ i)\, I(class|outcome\ i) = -\sum_{i=1}^{T} \frac{n_{i,\cdot}}{n_{\cdot,\cdot}} \sum_{j=1}^{C} \frac{n_{i,j}}{n_{i,\cdot}} \log \frac{n_{i,j}}{n_{i,\cdot}} .$$

Gini index of diversity: minimize the risk involved when making predictions once having made the test [3]; this splitting rule is invoked in **tgen** with "-g":

$$G(class|test) = \sum_{i=1}^{T} Pr(outcome\ i)\, G(class|outcome\ i) = \sum_{i=1}^{T} \frac{n_{i,\cdot}}{n_{\cdot,\cdot}} \sum_{j=1}^{C} \frac{n_{i,j}}{n_{i,\cdot}} \left(1 - \frac{n_{i,j}}{n_{i,\cdot}}\right) .$$

The correspondence between the two can be seen because they differ only in the inner terms, which can be shown to be approximately equal,

$$\log \frac{n_{i,j}}{n_{i,\cdot}} = \log\left(1 - \left(1 - \frac{n_{i,j}}{n_{i,\cdot}}\right)\right) \approx \left(1 - \frac{n_{i,j}}{n_{i,\cdot}}\right) .$$

Notice that if the task is to evaluate which cut-point to use for a real valued attribute (should we use the cut-point 200 for TSH in Figure 3.1 or some other value), rather than construct the table afresh for each potential cut point, we can repeatedly update the existing table for a sequence of adjacent cut points. Two adjacent cut-points and necessary table modifications are given in Table 3.2. In this table, m_i denotes the number of examples from class c_i that have

	class c_1	class c_2	...	class c_C	total
attribute-value < *cut-point*	$n_{1,1}$	$n_{1,2}$	...	$n_{1,C}$	$n_{1,\cdot}$
attribute-value ≥ *cut-point*	$n_{2,1}$	$n_{2,2}$	...	$n_{2,C}$	$n_{2,\cdot}$
total	$n_{\cdot,1}$	$n_{\cdot,2}$	...	$n_{\cdot,C}$	$n_{\cdot,\cdot}$
attribute-value < *cut-point* + δ	$n_{1,1} + m_1$	$n_{1,2} + m_2$	...	$n_{1,C} + m_c$	$n_{1,\cdot} + m.$
attribute-value ≥ *cut-point* + δ	$n_{2,1} - m_1$	$n_{2,2} - m_2$	...	$n_{2,C} - m_c$	$n_{1,\cdot} - m.$
total	$n_{\cdot,1}$	$n_{\cdot,2}$	...	$n_{\cdot,T}$	$n_{\cdot,\cdot}$

Table 3.2: Modifying Tables for Adjacent Cut-points

cut-point ≤ *attribute-value* < *cut-point* + δ, and $m.$ denotes their sum. To drive this modification process efficiently, the following algorithm is used:

1. Sort the examples according to the value of the real valued attribute.
2. Sequence through the examples in order and use the modification process above to evaluate each potential cut-point

3.3.4 Pruning methods

Pruning is often considered to be the most important part of the tree building task. Most approaches work using estimates of error and attempt to find a pruned subtree of the grown tree that minimizes this error estimate. Because the trees have been grown to "fit the data", these error estimates are usually pretty coarse.

To explain these methods, a few new terms will have to be introduced.

Root: A root is the starting or parent node of the whole tree.

Pruned subtree: A pruned subtree of a tree is formed by turning some of the nodes in the tree into leaves. A pruned subtree must have the same root as the original tree. So the root of a tree is the smallest possible pruned subtree, and the tree itself is the largest possible pruned subtree.

Sample error estimate: An estimate of the prediction error for a decision tree can be found by pumping a sample, N examples, through the tree and counting the number of times E that the tree misclassifies the examples. The estimate of error for the tree is then $\frac{E}{N}$. When the sample used to form the estimate is independent of the sample used to grow the original tree, this gives a good estimate.

Standard error of estimate: The standard error is intended to be an estimate of the standard deviation of the estimate. Given a proportion p derived from N examples, the standard error is given by

$$\sqrt{\frac{p(1-p)}{N}} .$$

Because some error estimates are derived in an unusual way, this formula is not always appropriate. However, it is often used anyway because it is an estimate where no other might be available.

With these in hand, a few error estimates can now be introduced.

Resubstitution error estimate

The *resubstitution error estimate* for a tree T is a sample error estimate. But the sample used to estimate error is the sample that was used to grow the tree. Because the tree has been "grown to fit the sample", this is usually an underestimate and the standard error of the estimate is not appropriate.

When a tree has been pruned without smoothing (i.e., if **tprune** used either "-n" or "-B" but not the smoothing option "-b"), and α was very low (e.g. "-A0.0001" or similar was used), this estimate is reported by **tclass** in the entry:

```
Expected accuracy for tree 1 = ...
```

Naive Laplacian Error Estimate

The *naive Laplacian error estimate* is intended to correct problems with the resubstitution error estimate, but only does a poor job. Suppose there are L leaves of the tree T having $n_1, \ldots, n_L$ examples and $e_1, \ldots, e_L$ errors when the resubstitution estimate is calculated. The resubstitution estimate is given by

$$\frac{\sum_{i=1}^{L} e_i}{N}$$

whereas the naive Laplacian estimate takes the Laplacian error estimate at each node $\frac{e_i+(C-1)}{n_i+C}$ (C is the number of classes) and averages these using a Laplacian estimate of node probabilities to get the error estimate

$$\sum_{i=1}^{L} \frac{n_i}{N} \frac{e_i + (C-1)}{n_i + C} .$$

For large n_i, the two estimates become indistinguishable. Again, because the tree has been grown to fit the sample, assumptions under which Laplacian error estimates are applicable are violated, so these estimates again tend to underestimate error, but less so.

When a tree has been pruned without smoothing (i.e., if **tprune** used either "-n" or "-B" but not the smoothing option "-b"), and α was 1 (e.g. "-A1"), this estimate is reported by **tclass** in the entry:

```
Expected accuracy for tree 1 = ...
```

Cross-Validation Error Estimate

Cross-Validation (CV) error estimates are used to estimate the error for a tree growing method rather than a particular tree. The idea is that, rather than making use of a sample to build a tree and a further sample to test the tree, you can manufacture several pseudo-independent samples from the original sample and use these to get a better idea of the error of the tree growing method.

The general technique takes a tree growing method M and estimates error of the method as follows.

1. Split the original sample S into v like-size disjoint samples $S_1, \ldots, S_v$.
2. For $i = 1, \ldots, v$,
   (a) Build a tree using method M on the training set $S - S_i$.
   (b) Determine the sample error estimate R_i using the test set S_i.
3. Form the CV-error estimate as
$$\sum_{i=1}^{v} \frac{|S_i|}{|S|} R_i \, .$$

To calculate the standard error of this estimate, the usual standard error formula is used.

This error estimation technique is guaranteed to give good estimates as the sample sizes become large. Breiman *et al.* argue from their empirical studies that v should be set to about 10. Cross validation can also be used as a means of evaluating a tree building method on a test set using many different samples with independent test sets.

This estimate can be obtained by using `ttest` with the "-C" option.

Error Estimate Pruning

A simple pruning approach by Bratko and Niblett prunes a node to a leaf if the naive Laplacian error estimate for the leaf at the node is less than the naive Laplacian error estimate for the subtree at the node.

Cost Complexity Pruning

Several *cost complexity* pruning methods make use of the notion of cost complexity. This is a measure of the resubstitution error of a tree further penalized by the size of the tree. Its main use is for forming a sequence of increasing pruned subtrees of a tree T; root of T has a pruned subtree T_1, which has a pruned subtree T_2, ..., which has a pruned subtree T. Without introducing such a notion, there is no real way of progressively pruning a tree in an ordered manner.

Cost complexity at level α for tree T, $R_\alpha(T)$: This is the formula

$$R_\alpha(T) = \textit{substitution-error-estimate} + \alpha|\textit{leaves}(T)| \, ,$$

where $|\textit{leaves}(T)|$ denotes the number of leaves in the tree T. The substitution error estimate is usually computed on a test set. Once again, notice that this is an additive function of the nodes in the tree.

R_α-minimizing subtree of T: This is a pruned subtree T' of T such that all other pruned subtrees either have a greater cost complexity at level α or they have the same cost complexity and they have T' as a pruned subtree. The R_α-minimizing subtrees as α decreases form an increasing sequence of pruned subtrees. Finding the R_α-minimizing subtree uses a standard algorithm for finding a pruned subtree minimizing an additive function. This is obtained with the "-c" option to **tprune** and using α as the option argument.

Minimum errors subtree of T: This is the R_0-minimizing subtree of T, or the smallest pruned subtree of T having the least substitution error. This is obtained with the "-c0" option or the "-M" option to **tprune**.

With these definitions in hand, we are now in a position to describe two more pruning algorithms. The first a test set to estimate error and determine at which level cost complexity pruning should be done.

The *cost complexity pruning algorithm with test set* [3, p79,p309] uses cost complexity to give an easily computed nested sequence of pruned subtrees and a test set to give "honest" error estimates for the pruned subtrees.

1. Split the sample into two disjoint sets, a training set and a test set.
2. Grow a tree using the training set.
3. Find the minimum errors subtree for the test set and compute its substitution error estimate R_0 from the test set and the standard error of the estimate SE_0.
4. The pruned subtree is now the R_α-minimizing subtree at the maximum level of α so that the pruned subtree has a substitution error estimate from the test set of less than $R_0 + SE_0$.

Stopping after Step 3 computes the so-called 0-SE tree. Stopping after Step 4 computes the 1-SE tree. Both can be obtained using the "-c" option to **tgen**.

The *cost complexity pruning algorithm with cross validation* [3, p79,p309] uses cross validation to form test sets instead. This is identical except that cross validation is used to estimate error.

1. Choose v disjoint subsets of the sample for cross validation.
2. Find the level of α minimizing the CV-error estimate for the R_α-minimizing subtree (i.e., the tree building method is grow a tree and find the R_α minimizing subtree). Let R_0 be the error estimate and let SE_0 be its standard error for that level of α.
3. Find the maximum level of α so the CV-error estimate for the R_α-minimizing subtree is less than $R_0 + SE_0$.
4. The pruned subtree is now the R_α-minimizing subtree constructed on the full sample.

The so-called 0-SE rule is obtained by ignoring Step 3. Either method of pruning is obtained using the "-C" option to **tgen**.

3.3.5 Handling Unknown Attribute Values

A problem that occurs commonly in practice is where examples are incompletely specified in the sense that their class is given, but some attribute values remain undetermined or unknown. This is often referred to as the problem of *unknown values*.

There are two situations where this problem arises. The first is where we have unknown values in the training sample used to grow the tree. The problem arises when we are to choose a new test at a tree node. How do we treat those examples for which the test outcome is unknown: how do we evaluate the test and would we subsequently partition the examples? The second problem arises when we come to classify a new example. As we pass the example down through the tree to a leaf, what do we do if one of the test outcomes is unknown?

A precise strategy for using a class probability tree T to classify an example x having some unknown attribute values is to process the example down through several branches of the tree, weighted by the probability that the example could occur at that branch. This follows from the probability identity

$$Pr(c|x,T) = \sum_{l=1}^{L} Pr(c|l,T)Pr(l|x,T) ,$$

where $Pr(l|x,T)$ denotes the probability that the example x belongs in the l-th leaf of T, and $Pr(c|l,T)$ denotes the probability that the class is c given that the example belongs in the l-th leaf of T. If all attribute values of x are known, $Pr(l|x,T)$ is 1 for the leaf l to which the example x belongs, and 0 for all others. If some attribute values are not known, the unit probability may be distributed across several leaves to which the example could belong.

Quinlan has made an extensive empirical study of various suggested solutions to these problems of unknown values [32]. Some of the methods compared included ignoring examples with unknown values, filling in the missing values somehow, splitting an example into a set of fractional examples with unknown values filled in, or treating unknowns as a separate outcome for each test. Several of these strategies can be implemented with the "-U" option in **tgen** and **tclass**. Not surprisingly, the best approaches were those that worked in accord with the strategy given previously. One places an example with unknown outcome proportionally to each test outcome or branch during test evaluation, partitioning and subsequent classification. For example, when the counting table is built, an example with class c_1 whose test outcome is unknown has its unit weight distributed across several of the rows in the column headed c_1. This means the counts in the table will not necessarily be integer. During partitioning, an example with test outcome unknown may be passed down several branches but such that each branch only gets part of the unit weight of the example. Finally, the same is done when a new example is to be classified.

3.3.6 An Example of Growing a Tree

Consider an LED display as drawn in Figure 3.2. The LED represents digits 0–9. The display is faulty, each element has 10% noise applied independently of the other elements. This is the LED example popularized by Breiman, Friedman, Olshen and Stone [3].

The classification task is to predict the digit intended to be represented by a particular configuration of the display. The learning task is to learn a classifier from examples. The theoretical maximum prediction accuracy obtainable for the classification task is about 72.7%.

Table 3.3 gives a part of an LED data set used to grow 2 trees. The full sample used to grow trees given below has 100 examples. The first row reads "elements L2, L3, L4 and L6 are on and the remainder are off, and the digit 4 was being represented".

The tree in Figure 3.3 is formed by growing a tree to completion and then pruning using Quinlan's pessimistic pruning algorithm.

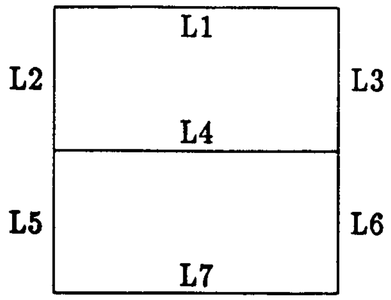


Figure 3.2: The LED display

digit	L1	L2	L3	L4	L5	L6	L7
4	n	y	y	y	n	y	n
8	y	y	y	y	y	y	n
6	n	y	n	y	y	y	y
6	y	y	n	y	y	y	y
7	y	n	y	y	n	y	n
2	y	n	y	y	y	n	y
1	n	n	y	n	n	y	n
9	y	n	y	y	y	y	y
2	y	n	n	y	y	y	y
5	y	y	n	y	n	y	y
2	y	n	y	y	y	n	y
9	y	y	y	y	n	y	y
3	y	n	y	y	n	y	y
7	y	y	y	n	n	y	n

Table 3.3: Part of a learning sample for the LED task

```
% mktree -p -en dig
% tprint -c dig.attr dig.treec
```

The tree has a true prediction accuracy of 71.0% and has 18 nodes. Each line represents a node in the tree. Non-leaf nodes give only a test outcome, while leaf nodes give a test outcome together with a vector of class counts. For instance, the first leaf in Figure 3.3 has "0+5+0+0+0+0+0+2+0+0" which indicates that the digit '1' occurred five times at the leaf, the digit '7' occurred twice, and all other digits no times.

The tree in Figure 3.4 is the right hand branch of a tree grown to completion. The full tree has true prediction accuracy of 68.2% and 69 nodes.

```
L5 = n:
|   L1 = n:
|   |   L4 = n: 0+5+0+0+0+0+0+2+0+0
|   |   L4 = y: 0+2+0+0+9+0+0+0+0+0
|   L1 = y:
|   |   L3 = n: 0+0+1+2+0+11+0+1+0+2
|   |   L3 = y:
|   |   |   L4 = n: 0+1+0+2+0+0+0+5+0+0
|   |   |   L4 = y:
|   |   |   |   L2 = n: 0+0+0+10+0+0+0+1+0+1
|   |   |   |   L2 = y: 0+0+0+0+0+0+0+0+0+8
L5 = y:
|   L2 = n: 1+0+12+0+0+0+0+0+0+1
|   L2 = y:
|   |   L4 = n: 7+0+0+0+1+0+0+0+0+1
|   |   L4 = y:
|   |   |   L3 = n: 0+0+0+0+0+0+6+0+0+1
|   |   |   L3 = y: 0+0+0+0+0+0+0+0+6+1
```

Figure 3.3: The pruned tree

3.4 A Guide to the Literature

The standard text from the statistics community is the CART book [3]. Good tutorial papers have been written by Hart [18], several by Michie [22, 21], Quinlan [29] and Quinlan *et al.* [33]. These all include descriptions of applications.

Many theses on trees exist, and some of the more recent ones are by Buntine [7], Catlett [13], Chou [15, 16], and Crawford [17].

Further interesting applications are; natural language speech recognition, where several complex methods were introduced to improve performance [1], and assessing credit cards [12, 23].

Experimental comparisons of all varieties exist. Some have compared general methods [43, 14], some compared components tree tasks such as pruning [31, 24, 26], splitting rules [25, 9], windowing [44], handling large data sets [13], missing or unknown attribute values [32], and comparisons between different theories [7, 5]. Always be wary about experimental evaluations because so many pitfalls exist.

Even a patent exists [36] on a tree growing method.

While it is difficult to survey in a short space the many issues that are of concern to the diverse research groups involved in tree learning, this section briefly reviews three broad issues that seem to be most important in extending tree methods. There is also, of course, the re-occurring issues of stopping, splitting and pruning rules, and treating unknown values.

3.4.1 Extended Representations

Trees are very verbose in representing certain disjunctive concepts. This was highlighted by Quinlan [31] who attempted to build a decision tree equivalent to the logical expression

$$A1 \wedge A2 \wedge A3 \ \vee \ A4 \wedge A5 \wedge A6 \ \vee \ A7 \wedge A8 \wedge A9 \, . \tag{3.1}$$

The smallest possible tree for this expression has 39 internal nodes and 40 leaves, considerably larger than the representation above, and contains considerable replication in that many different branches are identical in form. Most significant about this, a tree growing method will need to partition the data into 40 very fine partitions to grow the tree, whereas a disjunctive rule building method would (at best) only need to partition the data into 3 partitions, one for each conjunction. Since finer partitions give less accurate probability estimates, the tree growing methods are considerably disadvantaged on such problems.

There are many variations on this representation theme and several researchers have considered solutions. Quinlan has suggested post-processing trees into rule sets [30]. Matheus and Rendell, and Pagallo have proposed growing trees with conjunctive tests at nodes instead of single attribute tests [20, 27]. Chou has proposed an efficient algorithm for growing trellises instead of trees [16, 15] where trellises are directed acyclic graphs with class probability vectors at the leaves and tests at internal nodes (that is, like trees but internal nodes can have multiple parents). A related approach is the "pylons" of Bahl, Brown, de Souza and Mercer [1]. Smyth and Goodman [38] have developed an information theoretic approach to growing "non-directed" rules, that is, rules with many different attributes in the consequence. Weiss and Indurkhya [42] have developed a rule learning program that is a CART analog for rules.

An alternative to class probability trees for representing uncertain classification rules is the use of Bayes or causal nets [19, 28]. These essentially allow a modular decomposition of the attribute space using principles of independence, sometimes guided by intuition about causality. The simplest example is the simple (or "idiot") Bayes classifier which assumes all attributes are independent given class. These are highly competitive with trees on some problems [4, 14] and there is little doubt that with more thorough net learning approaches, this competitive performance can be considerably improved. Tree and rule learning methods could be incorporated in these approaches as methods for learning joint distributions at network nodes.

3.4.2 Extended Search and Alternate Growing Methods

The common framework of all tree growing methods discussed so far is recursive partitioning. This uses the simple one-ply lookahead strategy of growing a node to depth one for all possible test combinations and subsequently growing the tree according to the best test.

The question arises, can we do more extensive searches, for instance, a multi-ply lookahead? There is a problem with this in that lookahead when the number of examples are small, will rapidly cause performance to deteriorate because the best split may well be just a chance partition of the small sample into distinct classes. Weiss *et al.* [43] have demonstrated, in the context of learning conjunctive rules in noisy domains, that more extensive search can yield good results. To avoid the problem of small samples, they limit the size of potential rules.

3.4.3 Incremental Growing Methods

Some researchers have considered the question of how a learned tree can be efficiently updated, given new data. In the general framework, data arrives in a sequence and a corresponding sequence of trees is incrementally built. Early work here has been done in a noise-free context by Utgoff [39], and more sophisticated statistical approaches are suggested by Crawford [17]. See comments by Buntine in [5, 6].

3.4.4 Theoretical Developments

The theory of learning trees has been developed in several ways recently. Early statistical work was in an applied context, using techniques such as cost-complexity with cross-validation to do pruning [3]. The theory of minimum encoding, or MDL, has been applied by Quinlan and Rivest [34]. Proponents of the theory of minimum encoding have since developed this further. Rissanen includes a chapter in his book [35] and Wallace and Patrick have refined the theory and implemented a computer program [41]. Bayesian approaches similar to and extending these are developed in [7, 5].

3.4.5 Support for the Knowledge Engineer

An important issue raised in the introduction of this thesis is that learning of classification rules is typically performed as a service for the knowledge engineer. So a learning algorithm should not just propose a class probability tree suited for the classification task, it should provide any other information—maybe alternative class probability trees—that might assist the knowledge engineer with his duties.

Several groups have addressed this issue by making tree learning more interactive. A survey is given by Buntine and Stirling [10], and Shapiro's thesis also covers the topic [37].

Some forms of information that may be of use are:

Accuracy prediction: What kinds of accuracy (or more generally, performance) should be expected from the tree suggested?

Options: Are there any other trees that could just as well be used, and what is a measure of their suitability?

Ratings of tree components: At choice points in the tree building process, several options may be available, such as the choice of whether to prune a node and the choice of test to make at a new node. How good are these different choices?

Confidence: What is our confidence in any of the above predictions?

Various forms of these are developed and have been implemented in the IND package in the interactive interface and are based on the Bayesian theory.

```
L5 = n:
|   L1 = n:
|   |   L4 = n:
|   |   |   L6 = n: 0+0+0+0+0+0+0+1+0+0
|   |   |   L6 = y:
|   |   |   |   L2 = n:
|   |   |   |   |   L3 = n: 0+1+0+0+0+0+0+0+0+0
|   |   |   |   |   L3 = y: 0+3+0+0+0+0+0+1+0+0
|   |   |   |   L2 = y: 0+1+0+0+0+0+0+0+0+0
|   |   L4 = y:
|   |   |   L2 = n:
|   |   |   |   L6 = n: 0+1+0+0+0+0+0+0+0+0
|   |   |   |   L6 = y: 0+1+0+0+1+0+0+0+0+0
|   |   |   L2 = y: 0+0+0+0+8+0+0+0+0+0
|   L1 = y:
|   |   L3 = n:
|   |   |   L6 = n:
|   |   |   |   L2 = n:
|   |   |   |   |   L7 = n: 0+0+1+0+0+0+0+0+0+0
|   |   |   |   |   L7 = y: 0+0+0+0+0+0+0+0+0+1
|   |   |   |   L2 = y: 0+0+0+0+0+0+0+1+0+0
|   |   |   L6 = y:
|   |   |   |   L7 = n: 0+0+0+0+0+0+0+0+0+1
|   |   |   |   L7 = y:
|   |   |   |   |   L2 = n: 0+0+0+2+0+2+0+0+0+0
|   |   |   |   |   L2 = y: 0+0+0+0+0+9+0+0+0+0
|   |   L3 = y:
|   |   |   L4 = n:
|   |   |   |   L7 = n:
|   |   |   |   |   L2 = n: 0+1+0+0+0+0+0+2+0+0
|   |   |   |   |   L2 = y: 0+0+0+0+0+0+0+2+0+0
|   |   |   |   L7 = y:
|   |   |   |   |   L2 = n: 0+0+0+1+0+0+0+0+0+0
|   |   |   |   |   L2 = y: 0+0+0+1+0+0+0+1+0+0
|   |   |   L4 = y:
|   |   |   |   L2 = n:
|   |   |   |   |   L7 = n: 0+0+0+2+0+0+0+1+0+0
|   |   |   |   |   L7 = y:
|   |   |   |   |   |   L6 = n: 0+0+0+2+0+0+0+0+0+0
|   |   |   |   |   |   L6 = y: 0+0+0+6+0+0+0+0+0+1
|   |   |   |   L2 = y: 0+0+0+0+0+0+0+0+0+8
L5 = y:
    ...
```

Figure 3.4: Part of the unpruned tree

Chapter 4

IND Man Pages

4.1 Introduction

This chapter presents the man pages for IND. Man pages are included for every IND routine that is callable. Many IND routines, however, never need to be explicitly called by the user. Instead, these low-level routines are called by IND's higher-level routines. Because of this, it is usually not necessary to become familiar with all of the routines. We have included all of the man pages here for completeness, and also to help those who delve deeper into IND in an attempt to modify it.

The main man pages that the typical user of IND should be familiar with are: **attributes** (this is just a man page that describes the format of the attribute file—it is not the man page for an IND routine), **mkbld**, **mkclean**, **mktree**, **tclass**, **tgen** (because many of the options specified to **mktree** are explained in the **tgen** man page), **tprint**, and **tprune** (because many of the pruning options specified to **mktree** are explained in the **tprune** man page).

If you are interested in experiment control and design then you should like at **ttest** and the now out-of-date script **truns**.

If you are also interested in simple Bayes classifiers you should look at **mkcl**, **bclass**, and **bgen**, too.

4.2 The Man Pages

The man pages included in this chapter were printed with troff using commands such as "man -t mktree mkbld", etc., and just stuffed into the appropriate part of the manual. This was a lot easier than converting the pages to LaTeX, but unfortunately means that the man pages lack computer generated page numbers and might not be perfectly indexed.

NAME

attributes – describes the *attribute file* format for the IND family of programs

SYNOPSIS

none - not a program (just a man page)

DESCRIPTION

Overview

The *attribute file* (usually a file ending in ".attr") contains a series of attribute descriptions that guide IND's processing of examples. The examples themselves are not stored in the ".attr" file. The corpus of all examples available for the domain is usually stored in a file ending with ".dta". IND reads the examples in the ".dta" file and creates training and test sets of examples. Training data is usually placed in a ".bld" file ("bld" stands for "build") and test data is usually placed in a ".tst" file.

To make this concrete, consider the hypothyroid database in directory /IND/Data/thyroid. The file "hypo.attr" describes each attribute (including the class attribute) for the thyroid database. Each line in the "hypo.attr" describes one attribute. Attribute descriptions include the attribute name, attribute type, and allowable attribute values. The file "hypo.dta.Z" is a compressed version of all of the thyroid examples. Each line in the uncompressed version of this file is a single example containing a classification and a sequence of attribute values (in the order they are described in "hypo.attr"). IND routines sample the ".dta" file to create a training and test set (in "hypo.bld" and "hypo.tst", respectively). Other IND routines then build classifiers using the attribute descriptions in "hypo.attr" and the examples in "hypo.bld" and place these classifiers in either "hypo.tree" or "hypo.cl", depending on the type of classifier.

Attributes and Examples in More Detail

The attribute file contains a series of attribute descriptions separated by white space (space, tab, newline). Each attribute description contains an attribute name, followed by a colon, followed by a description of its type, and terminated with a full-stop (i.e., ".<CR>"). Any identifiers such as attribute names or attribute value names must be composed of letters, digits or the symbols "_-/.". The symbols ".∧" cannot appear in the first or last positions. For discrete attributes, the type description is a comma-separated list of attribute value names. For continuous attributes, the type description is a continuous type, *cont, step, norm,* followed by a range specifying the minimum and maximum value the attribute can take, represented as *min* .. *max.* The first attribute in the attribute file must describe the decision attribute.

The attribute file may have an optional *contexts* specification following attribute descriptions. Contexts are used to constrain the shape of the tree that can be generated by **tgen.** Contexts can also be used to prevent an attribute from ever appearing in a tree (using the form "never").

A context specification consists of the word "contexts" followed by a colon then a sequence of context descriptions for attributes. A context description is constructed from the following grammar:

```
context ::==    attribute-name "never" "." |
                attribute-name "onlyif" test "."
test ::==       literal |
                literal "and" ... literal |
                "or" ...  literal "and" ... literal
literal ::==    "(" test ")" | atom | "not" atom |
                "not" "(" test ")"
atom  ::==      attribute-name |
                attribute-name "=" attribute-value |
                attribute-name "<" attribute-value |
                attribute-name ">" attribute-value
```

These indicate that an attribute should be tested "never" or only if a certain condition holds. The test consisting of an attribute name alone holds if the attribute has itself been tested further up in the decision tree. Only the **tgen** program rigorously conforms to these specifications.

The attribute file may have an optional *utilities* specification. This specifies the utility "u(c,d)" of predicting class "c" when the true class is "d". This is specified by a matrix of comma delimited real values with each block delimited by semi-colons. An example specification for the three class case is:

```
  utilities : 100, 10, 10; 20, 50, 20; 10, 10, 100.
```

This means u(1,1) = 100, u(1,2)=10, u(2,1)=20, u(2,2)=50, etc. Utilities are taken into account by the *?class* programs when calculating the best decision. The default utility is "minimum errors", or a matrix with 1's on the diagonal and 0's elsewhere.

The attribute file is parsed with a yacc-generated parser, so does limited error reporting.

An example attribute file is shown below.

```
class:                  pass, fail.
Experience:             formal, repeating, self_taught, none.
Language:               assembly, basic, logo, none, other, pascal.
Sex:                    M,F.
HSC:                    cont 200..500.
Year:                   cont 60 .. 90.
MathsU:                 cont 0..4.
MarkM:                  cont    0 .. 200.
Faculty:                ARTS, ARTS/LAW, ECON, ECON/LAW, EDUC, ENG.
contexts:
        Sex       never.
        Language onlyif  not experience=none.
        MarkM    onlyif  MathsU > 0 or Faculty = ARTS.
```

The contexts here mean that the attribute Sex should never be tested, the attribute Langauge should only be tested if Experience has been tested previously to be something other than none, and MarkM should only be tested if MathsU has been tested to be >0 and if Faculty has been tested to be ARTS.

The example file contains input data (one record per line) matching the attribute description given in the attribute file. Fields are separated by tabs or spaces. Below is an example of input data matching the sml.attr attribute descriptions. Note that every entry in the file must contain a single example. In particular, this means that there cannot be any blank lines at the end of the file

```
fail    formal  assembly    M    289    82    3    100    ECON
pass    formal  assembly    F    357    81    2    81     ARTS
```

Other sample attribute files and examples databases can be found in "/IND/Data".

LIMITS

Currently, no discrete attribute or the class can have more the 32 values. If it does then the problem probably needs better engineering/structuring before applying IND.

The maximum number of attributes is 250. Again, if the problem has more than this, and they're all considered "possibly useful", then the problem probably needs a different non-tree system.

SEE ALSO

mkbld(1), *mktree*(1), *mkcl*(1), *bgen*(1), *tgen*(1).

NAME

bclass – classify a test set using a Bayes classifier

SYNOPSIS

bclass [options] *attribute_file cl_file(s)* [*test_file*]

DESCRIPTION

bclass takes a test set and classifies it according to the given Bayes classifier. The test set has the same format as the input data for **bgen** so decision attributes must be given (see *attributes*(1)).

If the attribute file contains a utilities specification, then average utility is also printed with any statistics, and the best decision is calculated to maximize expected utility. Otherwise the class with maximum probability is chosen (i.e., minimum errors utility).

OPTIONS

–b	Make summary of performance briefer. Useful if the output is later piped to a statistics program.
–c	Print the given class of each example.
–d	Print the decision made by the classifier on each example.
–o	Choose the best decision simply by picking the class with highest probability (i.e., ignore utilities).
–p	Print out the probability estimates for each class with each example.
–s	Print a summary of performance for the classifier: accuracy, mean square error, expected accuracy (the classifier's prediction of what accuracy it should have got) and an optional average utility.
–t	Print out the misclassification matrix of predicted classes by actual classes.
–u *p*	For the two class case, decide second class if probability of second class is greater than *p*. Default is 0.5. If utilities are specified in the description file, *p* is calculated automatically.

BUGS

Who knows? Hasn't been tested in quite a while.

SEE ALSO

attributes(1), *bgen*(1), *mkcl*(1).

NAME

bgen – generate a Bayes classifier

SYNOPSIS

bgen [options] *attribute_file examples_file cl_file*

bgen [options] *stem*

DESCRIPTION

bgen takes a file of examples and builds a Bayes Classifier. Real valued attributes are either given a normal model or a cut-point model, depending on whether the continuous type in the attribute file is **norm** or **step** or **cont** (see *attributes(1)).* The second form uses the stem instead of explicitly specifying the attribute, examples and output files. It assumes *stem.attr* and *stem.bld* exist. Creates *stem.cl.*

OPTIONS

–A *alpha*
Set alpha to value *alpha* (default is 0.5).

–n Force all real valued attributes to have normal models.

SEE ALSO

attributes(1), *bclass*(1), *mkcl*(1).

NAME

lstat – compute simple statistics on lines of data

SYNOPSIS

lstat [options] *files*

DESCRIPTION

lstat computes simple statistics on columns in files, such as column means, standard deviations, t-tests or F-tests and paired t-tests on a pair of columns. The first column is counted from 1. **lstat** is intended to be used on the ".trial" files output by **ttest**. The -T option makes certain changes that make output compatible with TeX tabular mode.

Either statistics are computed for a single file, or with the –F option, statistics are computed differentiating the first file from the remainder. When comparing columns, it is done on a line by line basis, so the paired t-test is used.

To check output from **ttest** for "hypo" files of training set size 1000, do

```
        lstat -v 1,2,3,5 hypo.trial.1000*
```

To do a paired t-test comparing the performance of "mdl" and "cart" trials (see *ttest(1)*) do

```
        lstat -F -s 1,2  hypo.trial.1000mdl hypo.trial.1000cart
```

OPTIONS

–2 Use a 2-tail t-test. Default is 1-tailed.

–a *col-list*
Print the mean of each column in *col-list*.

–A *col,col-list*
Print the mean of the difference between the *col*-th column and each column in *col-list*.

–C *col,col-list*
Print the mean, standard deviation, t-score and F-score of the difference between the *col*-th column and each column in *col-list*.

–d Be verbose during operation.

–e Suppress usual error reporting.

–f *prec* Floats are to be printed with *prec* decimal places after the point, default is 4.

–F Statistics are now computed as the difference between corresponding columns in the first file and subsequent files, with columns are paired as before for the paired t-test. In this mode, –s and –S options are identical, etc.

–n *text* In the output, precede the last field specified with a –s, –v, –V option, etc., with the text *text*.

–p *col-list*
The comma-separated list of integers, *col-list*, specifies which columns are to be printed as percents.

–q *items*
Only display this many items on a line of output.

–s *col-list*
Print the mean and t-score (difference from 0) of each column in *col-list*.

–S *col,col-list*
Print the mean and the t-score of the difference between the *col*-th column and each column in *col-list*.

–t *text* Seperate fields in the output with this text. For use with -T option.

–T *text* Make output compatible with TeX tabular mode. Damn useful for subsequently generating tables. *text* is printed before anything else. If a second –T option is used, then its *text* is printed right at the end.

–v *col-list*
: Print the mean and standard deviation of each column in *col-list*.

–V *col,col-list*
: Print the mean, and standard deviation of the difference between the *col*-th column and each column in *col-list*.

SEE ALSO

ttest(1).

NAME

mkbld – control partitioning of data

SYNOPSIS

mkbld [**–cDr**] [**–p** *i m*] [**–P** *i m*] *stem* [*size*] [*seed*]

DESCRIPTION

mkbld is a C shell script that runs **sample** and all sorts of clean up utilities in the process of building a training data set. *stem* should be a simple file name and not a path name. The script locates or builds a data file and then partitions that data file into a training set and a test set for use by the learning algorithms.

mkbld assumes files are stored in the format "*stem.ext*", where *stem* is the first argument to **mkbld** and the extension *ext* is one of:

".dta"	-	data set of all available examples for domain
".bld"	-	the training set or sample
".tst"	-	the test set
".all"	-	complete data (enumeration of all possible data)
".sh" -	shell script, outputs data to standard out	

The ".all" set is intended for data where an exhaustive enumeration of data points is available (e.g., for logical data sets such as XOR). It is incompatible with ".dta". The test set (".tst"), is normally the data set minus the training set, i.e., the remaining data, unless the data was obtained from a ".all" file, in which case the test set will be the entire ".all" file.

If the data file ".dta" doesn't exist, **mkbld** first tries to reconstruct it by running **uncompress**, and if that fails, by running the shell script "*stem*.sh" to generate it.

The second argument to **mkbld** tells how large the training set should be. It is not required when option -p or -P is used.

The optional third argument specifies a seed for the random number generator used in partitioning the data set. This allows trials to be reproduced later on.

OPTIONS

–c If a ".bld" file already exists, then report error and abort.

–D Echo the shell commands issued by **mkbld.** This is useful for debugging or learning how to use the system.

–p *m i* Split the data file into *m* partitions; use the *i*-th for
the test set, and the remaining *m*-1 partitions as the training set. Used to simulate cross validation on a data set via shell-level commands. Ignores the *size* parameter if specified.

–P *m i* Split the training file (the ".bld" file) into *m* partitions; use the *i*-th for the test set, and the remaining *m*-1 partitions as the training set. The new training set is placed in "stem.bld.i" where the suffix "*i*" indicates the partition *not* included in the file; the *i*-th partition is placed in "stem.tst". Used to simulate cross validation on a *sample* of a data set via shell-level commands. Ignores the *size* parameter if specified.

–r Do sampling with replacement (default is without replacement).

SEE ALSO

attributes(1), *mktree*(1), *mkcl*(1), *sample*(1).

NAME

mkcl – make and optionally test a Bayes classifier

SYNOPSIS

mkcl [**–D**] [**–c clopts**] [**–o genopts**] *stem*

DESCRIPTION

mkcl runs **bgen** and optionally **bclass** on the data set with file stem *stem*. This assumes an attribute file "*stem*.attr" exists, an examples file "*stem*.bld" exists, and, if **bclass** is to be run, that a test file "*stem*.tst" exists. The classifier is output to file "*stem*.cl".

OPTIONS

–c *clopts*
: Pass *clopts* argument as options to **bclass.** Note that *clopts* should be placed in quotes and must include it's own "-" (e.g., "-st").

–D
: Echo the shell commands issued by **mkcl.** This option is useful for debugging or learning how to use the system.

–o *genopts*
: Pass *genopts* argument as options to **bgen.** Note that *genopts* should be placed in quotes and must include it's own "-" (e.g., "-A 0.25 -n").

SEE ALSO

attributes(1), *mkbld*(1), *bgen*(1), *bclass*(1).

NAME

mkclean – clean up mess from **mkbld, mktree,** etc.

SYNOPSIS

mkclean [–crt] [*stem*]

DESCRIPTION

mkclean is a C shell script that cleans up any mess due to abnormal termination of **mkbld, mktree** or **ttest.** This means deleting any files "*stem*.bld", "*stem*.tst", "*stem*.tree*", and compressing the data file. *stem* should be a simple file name and not a path name.

OPTIONS

–c	Don't bother compressing data file.
–r	Tar and compress any "*stem*.trial.*" files.
–t	Don't delete tree or classifier files.

SEE ALSO

mkbld(1), *mktree*(1), *ttest*(1).

NAME

mktree – make a decision tree and optionally prune and test it

SYNOPSIS

mktree [**–a**] [**–c** *clopts*] [**–D**] [**–n** *tname*] [**–o** *genopts*] [**–p** *propts*] [**–R** *r n*] *stem*

DESCRIPTION

mktree is a C shell script that runs **tgen** and optionally **tprune** and **tclass** on the data set with file stem *stem*. This assumes an attribute file "*stem*.attr" exists and a training data file "*stem*.bld" exists, presumably produced by **mkbld**.

Pruned trees are output to file "*stem*.tree" if only a single tree is produced, or for multiple trees, to "*stem*.tree.1, ... *stem*.tree.n". The original unpruned trees will be in "*stem*.treec", etc.

A typical command sequence to run trials on the "hypo" data set might be:

```
# select a sample of 500 data points
mkbld hypo 500
# run tgen with "-tuU1 -d5", tprune with "-b",
# and tclass with "-slvg"
mktree -o "-tuU1 -d5" -p "-b" -c "-slvg" hypo
```

Some common forms are:

```
# CART-like: GINI splits and cost-complexity pruning
#      using 10-fold cross-valid. and 0-SE rule
mktree -o "-gC10 -p0 -A0.1" -p "-n" -c "-slv" stem

# C4-like: info. gain splits, pessimistic pruning
mktree -o "-uU1 -A0.1" -p "-en" -c "-slv" stem

# MDL-like: Bayes splits and coding of tree
mktree -o "-uU1 -A1 -NP-0.69315,-0.69315,02" \
      -p "-B" -c "-slvgQ" stem

# averaging many trees using options facility;
#      take care with depth, set bounds and alpha (-A)
mktree -o "-tuU1 -B2,3 -J3 -K3 -d5 -s3" -p "-b" -c "-slvgQ" stem
```

Note that arguments to be passed to **tgen**, **tprune**, and **tclass** should be enclosed in quotes.

OPTIONS

–a Output character format for trees as well using **tchar.**

–c *clopts*
Run **tclass** on all trees with options *clopts*, with output to standard output. The argument *clopts* should be enclosed in quotes. See *tclass*(1) for **tclass** options.

–D Echo the shell commands issued by **mktree.** This option is useful for debugging or learning how to use the system.

–i *nice* Automatically nices everything (see the C shell "nice" command) to this value. Default is 10.

–m *datasize*
Never let any program use more than this memory. Only usually a problem with the –J option in **tgen**. See "limit datasize" in **csh**. Default is 12Mb (i.e., "-m 12m" or "-m 12000k").

–n *tname*

Place tree in file "*stem*.tree*tname* ("*stem*.tree" followed by *tname*).

–o *genopts*
: Pass argument *genopts* as options to **tgen.** Multiple flags will build multiple trees numbered 1, 2, etc. The argument *genopts* should be enclosed in quotes. See *tgen*(1) for **tgen** options.

–p *propts*
: Run **tprune** with options *propts* on all trees. The argument *propts* should be enclosed in quotes. See *tprune*(1) for **tprune** options.

–R *r n*
: Make (*n*-1) trees after the first tree using the "-R *r*" option to **tgen,** as well as any options mentioned in the –o option (multiple –o options should not be used). This option is currently unuseable.

–t *cputime*
: Never let any program use more than this many seconds. (See "limit cputime" in the C shell) Default is 2000 seconds (i.e. "-t 2000").

SEE ALSO

attributes(1), *mkbld*(1), *tchar*(1), *tclass*(1), *tgen*(1), *tprune*(1), *ttest*(1).

NAME

sample – randomly sample lines from a file

SYNOPSIS

sample *nlines* [**-r**] [**-N** *Nlines*] [**-o** *output*] [**-p** *p,i,name* [**-s** *seed*] [**-t** *rejects*] *files*

DESCRIPTION

sample does random sampling of lines in a file, with or without replacement. It also allows optional collection of the unsampled lines for use as a test file. Users of **IND** usually do not need to call **sample** directly. Instead, **mkbld** (which then calls **sample** for you) should be used.

OPTIONS

–r Sample with replacement. The default is without.

–N *n* If *n*, the size, of the input file is known, the -N option makes the sampling algorithm more efficient in space and time.

–o *file* Put output into *tfile*, otherwise output goes to stdout.

–p *p,i,name*
Partition the file into *p* equal pieces, no random sampling. Place the *i*-th partition in the file *tname* and the rest go to the usual sample output. Used for cross-validation with *i* in 1,...,*p*. The first argument *nlines* is ignored with this option but some integer still needs to be present.

–s *x* Use *x* as a seed to the random number generator.

–t *file* Put rejects into *file* (if the -r option isn't used), otherwise rejects are discarded. Rejects are lines not included in the sample.

SEE ALSO

mkbld(1).

NAME

tchar – convert a tree to or from character format

SYNOPSIS

tchar [**–a**] *attribute_file treein treeout*

DESCRIPTION

tchar converts a tree from usual binary format to character format. This can be useful for manual pruning or alterations to the tree. Convert the tree to character form, edit, then convert back to binary form. It can also be used for creating input to **tgendta,** or for transferring trees to another computer.

Each line contains an indented node description. Lines are in printed in a different order to **tprint** (with a right to left pre-order traversal instead of a left to right). Fields are: node type, class counts for examples at that node for each class followed by the total count, and leaf probability in parenthesis. Test nodes have additional node-flags (see data structures), and a representation of the test made (cuts begin with 6, discrete splits with 10). Option trees are more complicated again.

For example, given the tree below, grown using "mktree -o '-tA1 -P0,-0.7,02' hypo" and printed using "tprint -cd hypo.attr hypo.tree",

```
TSH < 6.05: +0+318+0+0 negative
TSH >= 6.05:
|  TSH_measured = f: +0+36+0+0 negative
|  TSH_measured = t:
|  |  FTI < 64:
|  |  |  query_hypothyroid = f:
|  |  |  |  T4 < 1.45: +0+0+13+0 primary_hypothyroid
|  |  |  |  T4 >= 1.45: +0+1+0+0 negative
|  |  |  query_hypothyroid = t: +0+1+1+0 primary_hypothyroid
|  |  FTI >= 64:
|  |  |  on_thyroxine = f:
|  |  |  |  T4_measured = f: +0+0+1+0 primary_hypothyroid
|  |  |  |  T4_measured = t: +23+0+0+0 compensated_hypothyroid
|  |  |  on_thyroxine = t: +0+6+0+0 negative
```

and running "tchar hypo.attr hypo.tree hypo.ctr ; cat hypo.ctr" we get the following:

```
SIZE: 15 8 0 0
PRIOR: 1 0 -0.7 2 0 20
1, 23+362+15+0 = 400 (0) 20 6 18 6.05 (0.000000 0.000000)
1, 23+44+15+0 = 82 (0) 20 10 17 (0.000000 0.000000)
 1, 23+8+15+0 = 46 (0) 20 6 26 64 (0.000000 0.000000)
  1, 23+6+1+0 = 30 (0) 20 10 3 (0.000000 0.000000)
   2, 0+6+0+0 = 6 (0)
   1, 23+0+1+0 = 24 (0) 20 10 23 (0.000000 0.000000)
    2, 23+0+0+0 = 23 (0)
    2, 0+0+1+0 = 1 (0)
  1, 0+2+14+0 = 16 (0) 20 10 10 (0.000000 0.000000)
   2, 0+1+1+0 = 2 (0)
   1, 0+1+13+0 = 14 (0) 20 6 24 1.45 (0.000000 0.000000)
    2, 0+1+0+0 = 1 (0)
    2, 0+0+13+0 = 13 (0)
 2, 0+36+0+0 = 36 (0)
2, 0+318+0+0 = 318 (0)
```

Notice the node and leaf counts respectively are given after "SIZE", details of the original prior specified with the −A and −P options are given after PRIOR and each line following represents a node or leaf. Test nodes begin with 1 and leaf nodes with 2. The first line of counts indicate the training set had 400 examples, 362 of the most common class. The first test on "TSH" has attribute number 18, "negative" has class value number 1, etc.

OPTIONS

−a Converts in the reverse direction (from character to binary).

−c If its a tree with counts instead of probabilities, assume the leaf counts are correct and total all other counts and totals accordingly.

SEE ALSO

tgendta(1), *tprint*(1).

NAME

tclass – classify a test set using a decision tree

SYNOPSIS

tclass [options] *attribute_file tree_file(s)* [*test_file*]

tclass [options] *stem*

DESCRIPTION

tclass takes a test set and classifies it according to the given tree(s). The test set has the same format as the input data for **tgen** so a decision attribute must be given. The second form uses the stem instead of explicitly specifying the attribute, examples and output files and assumes files "*stem*.attr", "*stem*.bld" and "*stem*.tree" exist.

If multiple *n* trees are input, the final class probability vectors from each tree are merged (by default, averaged) to give an *(n+1)*-th prediction. This is termed the "multiple tree". Statistics can be reported for both the individual trees and/or the multiple tree. The facilities in **tgen** for generating multiple trees are currently under repair so this option is currently not useable.

If the *attribute_file* contains a utilities specification, then average utility is also printed with any statistics and the best decision is calculated to maximize expected utility. Otherwise, the class with maximum probability is chosen (i.e., minimum errors utility).

The usual output mode for **tclass** is fairly verbose. Output will be brief and restricted to a single line with the –b option. The order of output in this case is the same as for verbose mode except that no explanation, etc. will be given, simply a line of numbers. The order for a single tree is as follows: percentage correct, half-brier score, predicted percentage correct (with –s option), standard deviation of prediction (with –v option), log posterior probability (with –g option), node count and expected number of nodes (with –l option), utility on training sample (if utilities exist in *attribute_file*.), and if the tree was grown with cross validation, the cross validation estimate of percentage accuracy and its standard deviation (with –G option).

OPTIONS

–A *alpha*
: Same options as for tree prior as in **tgen**.

–b Make summary of performance briefer. Useful if the output is later piped to a statistics program.

–c Print for each example the given class. This option combines nicely with option –d.

–D Print for each example the decision for each tree (use with the –m option).

–d Print for each example the decision for the single tree, or if several trees exist, the multiple tree. This option combines nicely with option –c.

–e Print for each example and for each tree whether the decision agreed with the actual class (1=yes, 0=no). Useful for comparative statistical analysis of tree accuracy.

–g Print out the posterior for the trees as well (taken from the header).

–G Assuming the tree was generated using the –C option of **tgen** using cost-complexity pruning, this prints out the error estimate for the tree calculated during the cross validation procedure. Only works if the tree was pruned with the –n option rather than Bayes pruning. Notice the estimate will be biased because the cost-complexity tradeoff parameter was selected to minimise errors.

–l Print out the leafcount (both the expected and actual sizes) for each tree.

–m *n* Classify multiple trees, where *n* is the number of trees. The *n* trees are listed in the *tree_file*(s) argument. If the –m option is not used, one tree is assumed.

–o Choose best decision simply by picking the class with highest probability (i.e. ignore utilities).

–P For each example, print out the probability estimates for each class and for each tree (use with the –m option).

–p For each example, print out the probability estimates for each class for the single tree, or if several trees exist, the multiple tree. When coupled with the –v option prints their variance as well.

–Q Print out details of the tree prior (assumed constant across multiple trees) once at the beginning.

–q When using multiple trees, prints out a matrix representing differences between trees. The second row and column of the matrix represents the differences of other trees with the second tree, etc. (the last row and column, the multiple tree). Useful for determining which single tree is most similar to the multiple tree. Differences are measured in terms of the proportion of examples on whose classification two trees disagree (in the upper triagonal) and the average over the examples of the manhattan distance between class probability vectors produced by two trees on an example (in the lower triagonal).

–S Print out a summary of performance for each tree (use with the –m option).

–s Print a summary of performance for the single tree, or if several trees exist, the multiple tree. This includes accuracy, mean square error, expected accuracy (the classifier's prediction of what accuracy it should have got, found by averaging the class probabilities at the leaves) and an optional average utility. Expected accuracy is usually an over estimate, except in the case of a small tree with "lots" of data, or in the case of an option tree built using the –J option to **tgen** and realistic prior parameters.

–t Print out with the other statistics a misclassification matrix of predicted classes by actual classes for trees (either the –s or the –S options must be used).

–U *n* How to handle unknowns when classifying. The methods available are:

- 1 Send the unknown down each branch with proportion as found in the training set at that node.
- 3 Send unknown down the most common branch (the default).
- 4 Send the unknown down a single branch chosen with probability proportional to that found in the training set at that node.

–v For each example, a variance of the expected accuracy is calculated. The average of these variances is then printed. Unfortunately, this is not a variance for the expected accuracy of the sample (this much more complicated formula is not calculated), but the value gives a generous over-estimate of the imprecision in the expected accuracy of the sample.

–W *"w1 ... wm"*
When averaging class probabilities from multiple trees, weight the i-th tree by the weight *wi* (only use with the –m option). Weights are white space delimited (so the argument must be enclosed in quotes). To construct the argument automatically, **thead** and an **awk**-like program may be useful.

–Z By default, leaf nodes which have zero count are assigned the same class probabilities as their parent. With this flag set, zero-count nodes are assigned the class probabilities found at the root of the tree.

SEE ALSO

tgen(1), *tprune*(1), *mktree*(1), *ttest*(1).

NAME

tgen – generate a decision tree

SYNOPSIS

tgen [options] *attribute_file example_file tree_file*

DESCRIPTION

tgen takes a data set (see *attributes*(1) for a description of the data file formats and */IND/Data/thyroid* for a sample data set) and builds a decision tree. Options allow CART style cost-complexity pruning by test set or by cross validation, and a wide variety of splitting rules such as Bayesian, information gain and GINI methods. Subsetting is implemented. Various hacks exist for handling missing values. Lookahead can be programmed with the –B option, and early stopping (pre-pruning) with the –J1 option. Interactive mode (the –o option) also displays graphs under X.11 of the cut-point profiles, if you wish to control the growing operation more closely.

The Bayesian option trees for averaging is started by combining the –B, –J, and –K options. This is in development stage, and is a simplistic search that requires large amounts memory and time, so it may have to be nursed. See option descriptions and bugs. The –B option allows n-ply lookahead during splitting (all other splitting rules use 1-ply lookahead). Use 2 or 3-ply to get better performance on small problems, or combine with the –bq2 option in **tprune** to get even more sophisticated search for the single best tree. Option trees are initiated with the –J option, and are best combined with solid stopping rules such as a depth bound (the –d option), and the set size bound (the –s option). The –K option is for post-pruning of option trees only. A typical option combination might be:

```
tgen -t -B2,4  -J4 -K4  -d5  -s4 ...
```

Appropriate depth and set size bound should be chosen with the application in mind.

OPTIONS

–a Write out tree in character format instead of usual binary.

–A *alpha*

Probabilities at leaf nodes are calculated using the Laplace formula:

(#this-class + *alpha*/#classes)/(#total + *alpha*).

where

#this-class = count for this class at this node
#total = total count at this node
#classes = number of classes

Note the class frequencies sum to 1. The default is *alpha*=1. This flag also effects the operation of the *–t* flag because *alpha* is used as a prior parameter. See also the –P options.

–B *depth,breadth[,fact]*

When tree growing there is an initial beam-search n-ply lookahead phase to evaluate the quality of each test. At each step when doing this, choose the best *breadth* choices for each test that are within *fact* of the best, and add these as options on the search beam. Lookahead to depth *depth*. Only supported with the –t option. Default values are 1,1,0.00001.

–c *prop* Build tree from a proportion *prop* of the examples selected at random; prune tree using cost-complexity pruning with test set on remainder. A typical value to use is 0.7. See the –G option in **tclass** for displaying the error estimate, and the –p option below for setting the standard errors.

–C *folds*

Build tree using *folds*-fold cross-validation cost-complexity pruning. CART recommended value is 10-fold. See the –G option in **tclass** for displaying the error estimate, and the –p option below for setting the standard errors. A second –C *folds* on the command line will have **tgen** report additional information calculated during the pruning operation.

–d *depth*
Stops building tree after depth *depth*. By default is set to number of attributes plus twice the number of continuous attributes.

–g Use GINI index of diversity when splitting.

–J *breadth*[*,fact*[*,add_fact*[*,leaf_fact*]]]
Does option tree growing with magic numbers to alter the search strategy, and requires use of the –B option (at the very least, –B1,2). After initial lookahead has found a candidate set of tests nodes, grow as distinct optional sub-trees the best *breadth* test nodes within *fact* (default = 0.005) factor of the best. The last two magic numbers modulate early stopping or pre-pruning. Only grow the node if the non-leaf probability is within a factor *leaf_fact* of leaf probability (default = 0.00001, make this closer to 1.0 to stop earlier) and if the non-leaf probability is not greater than a factor *add_fact* of the best test to grow (default=0.75, make this smaller to stop earlier). This option is only supported with the –t option. Should be used with options –d and –s to help limit search and option –K to save memory.

–K *breadth[,fact]*
Does post-pruning on option trees with magic numbers to alter the search. Keep only the best *breadth* (default = 1) option branches and only choose those within a *fact* (default = 0.005) factor of the best. Only supported with the –t option.

–M Marshall modification to gain.

–N When using the –t or –J options the Bayes splitting rules, etc., are in effect. For these, a "log posterior" measure is computed and used as a rating for the tree (see –g option in **tclass**). This is usally not quite correct in that the tree prior, as specified with the –P option has not been normalized. The –N option does the extra calculation necessary to compute this normalizing constant, which can then be displayed with the –Q option in **tclass**. The computation can be exponential in nature if there are mixed continuous or multi-valued attributes. The calculation is incorrect if subsetting is used. Help avoid this with the –d option, for instance, try smaller depths first.

–o Manual override flag. Allows the user to manually choose which attribute to split on, and print all sorts of debugging information while building, thus overriding the automatic selection made. A menu of interactive options is available (via the "h" command) to guide the manual tree building process. Setting the "x" toggle can spawn **xgraph** processes giving cut point profiles. These may have to be killed manually. This option isn't supported with the –B, –J or –K options.

–p *factor*
When cost-complexity pruning, number of standard deviations to use. Default is 1.0. CART recommends 0.0 for larger trees and (sometimes) greater accuracy.

–P *n-weight,l_weight*[*,oflags*]
This option sets tree prior parameters node-weight and leaf-weight (the log-prior for these nodes in a tree). Only the first parameter is essential. If the 02 bit is set in *oflags*, modify the node weight by subtracting the log of the number of test choices at that node.

–r *octal-flags*
Print tree at the end using the octal coded print flags *octal-flags* interpreted as described in the header file /IND/Tree/TREE.h. Useful mainly for debug.

–s *min* Turn node to leaf (stop growing) if examples less than *min*.

–S *type* Allow binary tests on multi-valued discrete attributes which split the attribute values into two parts. This is "subsetting" implemented in a simple greedy manner. *type* can be one of the following:

00 Regular subsetting of multi-valued attributes. i.e. do splits testing if the attribute is in a certain subset or not.

02 Do binary encoding of the multi-valued attributes. i.e. do splits testing if the attribute is a certain value or not.

–t Bayes splitting rule.

–u Apportion unknown values when evaluating splits.

–U *n* How to handle unknowns when splitting training set. The available methods are:

1 The default. Send the unknown down each branch with proportion as found in the training set at that node. Not yet convinced the implementation is OK.

3 Send unknown down the most common branch.

4 Send unknown down a single branch chosen with probability proportional to that found in the training set at that node.

–W *cycles[,alphamin]*
Do a trick suggested by Wallace and Patrick to determine the best value of *alpha* (the parameter passed to the –A option). Grow a tree (or option tree) with the initial value of *alpha*. Then adjust *alpha* so that the posterior probability for the tree (for instance, as printed using the –g option to **tclass**) is at a local maximum for *alpha*. Now grow a tree again using the new value of *alpha*. Repeat until you've done *cycles* cycles or *alpha* has changed no more than accuracy 0.01 from the last cycle. In addition, *alpha* is prevented from going below *alphamin*. A good cycle maximum *cycles* would be 4, so that at most 4 trees are grown. When using the –J option, because of time, it would be better to use *cycles*=1. A good value for *alphamin* is 1.0 if you expect high accuracy, and more if you expect less accuracy.

–Z By default, leaf nodes which have zero count are assigned the same class probabilities as their parent. With this flag set, zero-count nodes are assigned the class probabilities found at the root of the tree.

BUGS

If **tgen** quits with a message like "memory limit exceeded" or "time limit exceeded" then it still produces a tree, but has stopped search prematurely. The tree may have been grown in a lob-sided manner so the performance of the tree may be very poor. One can extend time or memory limits using **limit** (see the –t and –m options in **mktree**), or decrease the search by decreasing the depth, breadth or factors in the –B, –J or –K options. It is always useful to check the "log posterior" of the tree using the –g option to **tclass** or the –s option to **thead**, to see if it is smaller than the log posterior for a tree produced without the –J option. If the –J tree's is smaller, it is probably a lob-sided tree and will perform poorly. Likewise for "expected leaf count".

When using **xgraph** to display cut-point profiles, you will have to kill the **xgraph** processes yourself.

Probably lots more.

SEE ALSO

mkbld(1), *tprune*(1), *tclass*(1), *mktree*(1), *thead*(1), *tprint*(1).

NAME

tgendta – generate data to match decision tree

SYNOPSIS

tgendta [**–afp**] [**–i** *spacing*] [**–n** *examples*] [**–s** *seed*] *attribute_file tree*

DESCRIPTION

tgendta generates data randomly from the decision tree and outputs to stderr. The decision tree is either assumed to be a class probability tree specifying a probability distribution, or, with the –p option, a logical specification.

OPTIONS

–a Input a tree in character format.

–f Generate all possible examples (or at least a representative set if real values exist). This option doesn't make sense without the –p option.

–i *spacing*
With the –f option, says that roughly *spacing* examples will be generated, equally spaced, when filling in values of real valued attribute. Otherwise, two different values will be given at each leaf.

–n *examples*
Number of examples to generate. Ignored with the –f option.

–p The data (and presumably the decision tree) has no noise.

–s *seed* Seed to initialize the random number generator.

SEE ALSO

tchar(1).

BUGS

Handling of real values is a hack.

NAME

thead – print details about a decision tree

SYNOPSIS

thead [options] [**–m** *n*] *tree* ...

DESCRIPTION

thead prints details about tree(s) built by **tgen**. The brief options are useful when piped through awk-like tools to the –W option in **tclass.**

OPTIONS

–A *alpha*
: Same options as for tree prior as in **tgen.**

–l Print number of leaf nodes on one line without verbage.

–m *ntrees*
: Indicates how many trees, if more than one.

–P *opts* Same options as for tree prior as in **tgen.**

–p Print information about the prior structure stored for the tree (alpha, etc.).

–s Print leaf count, nodes, and the "weight" (the "sprob" field) which is the log probability for a Bayes tree.

–t Print number of nodes on one line without verbage.

SEE ALSO

tgen(1), *tprune*(1), *tclassify*(1), *tprint*(1).

NAME

tprint – print a decision tree

SYNOPSIS

tprint [**options**] *attribute_file tree_file*

tprint [**options**] *stem*

DESCRIPTION

tprint displays a decision tree built by **tgen**. The second form uses the *stem* instead of explicitly specifying the attribute, examples and output files. It assumes "*stem*.attr" and "*stem*.tree" exist.

Each line contains a test on an attribute value pair. If the test leads to a leaf, information about the leaf is printed at the end of the line. If the test leads to a subtree, the subtree is printed (indented four spaces) below the test.

Using the –t option, a test set can be run through the data to display, at a glance, how the tree classifies the test set. Useful for finding out where in the tree is making the most errors.

Various other options allow details of internal nodes to be printed, leaf posteriors, standard deviations of probabilities, and classification probabilities (in an averaged tree, leaf probabilities usually differ from final probabilities for examples at that leaf).

If the tree is currently in counts form, and the printing options you specify require it to be in probability form, then an appropriate conversion will be done.

OPTIONS

–A Same options as for tree prior as in **tgen**.

–a Read tree in in character format (produced by **tchar**).

–b By default, probabilities displayed using the **–p** option are for those at the node. This option displays the final probabilities that would be used by the classification routine, after all tree averaging has been done.

–c Display counts of training set examples in each class. The counts are printed out in the same order as the classes appear in the attribute file.

–D *n* Only print out tree to depth *n*.

–d Display the best class (the decision). Takes account of utilities or cut-off probabilities.

–E Same options as for tree prior as in **tgen**.

–i Display counts, etc., for interior (non-leaf) nodes as well.

–p Display proportion of each class (number of training set examples in that class divided by the total number of training set examples at the node). The proportions are printed out in the same order as the classes appear in the attribute file. If flagged twice, then display standard deviations as well.

–P Same options as for tree prior as in **tgen**.

–q Display posteriors used by tree averaging routine at each node. At each leaf node, L labels the posterior probability of that leaf node being in a tree. For options, P labels the posterior probability of that option being the test occurring in a tree.

–t *test_file*
This processes a test file *test_file* as would normally be done by tclass to produce vectors of class counts at each node. These are printed as for the –c option. The method of handling unknowns can be set using the –U option.

–U *n* How to handle unknowns when classifying. The available methods are:

1 Send the unknown down each branch with proportion as found in the training set at that node.

3 Send unknown down the most common branch (the default).

4 Send the unknown down a single branch chosen with probability proportional to that found in the training set at that node.

-Z By default, leaf nodes which have zero count are assigned the same class probabilities as their parent. With this flag set, zero-count nodes are assigned the class probabilities found at the root of the tree.

SEE ALSO

tgen(1), *thead*(1), *tclass*(1), *tprune*(1).

NAME

tprune – prune a decision tree

SYNOPSIS

tprune [options] *attribute_file tree*

DESCRIPTION

tprune simplifies a decision tree by removing (or pruning) subtrees, and then converts the counts in the nodes to probabilities. Flexible combinations of the different pruning algorithms are available. Can use depth-bounded pruning, with cost-complexity or pessimistic or minimum errors pruning. Option trees, however, can only be pruned in a depth-bounded manner. This is then followed by count to probability conversion with or without Bayesian tree smoothing. The pruned tree is written to the file "*tree*.p".

OPTIONS

–A Same options as for tree prior as in **tgen**.

–D Prune node if all subtrees make the same decision. Done after everything else.

–b Convert counts to probabilities using Laplacian estimates, and install leaf probabilities for Bayesian tree smoothing later by **tclass**.

–B Like the –b option but picks the best pruned subtree and gives all its leaves a leaf probability of 1. This corresponds to doing minimum encoding (MDL, MML) pruning because it prevents later tree smoothing.

–c *factor*
Do cost-complexity pruning with trade-off set by *factor*. See also the –V option.

–d *depth*
Before other pruning methods, strip everything below *depth* depth.

–e Pessimistic pruning, one interpretation.

–E Same options as for tree prior as in **tgen**.

–M Prune to minimum errors subtree.

–n Convert counts to probabilities using Laplacian estimates, and make all leaves have a leaf probability of 1, to prevent subsequent Bayesian smoothing by **tclass**.

–o *options*
When tree smoothing, prune node to leaf if it has more than *options* options. A good default value to use is 10.

–p *factor*
Set prune factor. The pessimistic pruning algorithm prunes a subtree if its error is within *factor* standard errors of a pessimistic estimate of the error. The default factor (when doing pessimistic pruning without specifying a factor) is 1.0. Be sure to also use option –e when using this option.

–P Same options as for tree prior as in **tgen**.

–q *factor*
Set prune factor. When tree smoothing (option –b), remove any option branches whose proportion is less than this value. Default is 0.01. Setting factor to value greater than 1 has the effect that all option branches other than the best are pruned (this is similar to the –B flag applied to option trees, but the tree is still able to be smoothed afterwards).

–r *octal-flags*
Print tree at the end using the octal coded print flags *octal-flags* interpreted as described in the header file /IND/Tree/TREE.h. Useful mainly for debug.

–V *testfile*
Use *testfile* to determine trade-off for cost complexity pruning. Default standard errors is 1.0.

Preceed this option with the –c option if you wish to set the standard errors to something else.

SEE ALSO

tgen(1), *tclass*(1), *thead*(1).

NAME

ttest – build and test trees and report statistics

SYNOPSIS

ttest [options] *stem size [genopts]*

DESCRIPTION

ttest is a **csh** script used to control the running of experiments on trees. *stem* should be a simple file name and not a path name. A sequence of training/tests pairs are generated using **mkbld**, various trees are built and tested on these using **mktree** with different options, and statistics can be collected in separate report files or output to stdio using **tclass**. A final summary report is output to stdio using **lstat**. *stem* is the data set stem to use, *size* is the size of training sets to generate and the optional argument *genopts* are default options always passed to **tgen** when generating trees. *genopts* is by default "-uU 3".

Control of which tree generation and prune combinations to use is specified by the –T and –R options. A typical command sequence to run MDL-like and CART-like trials on the "hypo" data set might be:

```
ttest -T "-uU1#-tP-.7,-.7,02#-A0.3" "-B" "mdl" \
     -T "-uU1#-A0.01#-gC10#-p0" "#-n" "cart" \
     -c -sblv hypo 500
```

See the description of the –T option, below, to interpret this. The "#"s are replaced by white space before being passed to **mktree.** With this command, **ttest** will first output the following summary,

```
Running trials:
    tgen -uU1 -tP-.7,-.7,02 -A1 hypo... ; tprune -B
    tgen -uU1 -gC10 -p0 hypo... ; tprune -n
Redirecting results to:
    hypo.trial.500mdl
    hypo.trial.500cart
```

and then proceed to run the trials indicated using files hypo.bld, hypo.tree.1, etc. In this case, for each tree generated, **tclass** is run using options "-sblv" and the output appended to the respective report files.

Selection of training/test data pairs is controlled by the –C, –V and –v options. This allows cross validation, random generation of partitions according to a list of seeds, or cross validation on random partitions.

OPTIONS

–c *clopts*

Pass these options to **tclass** when generating statistics on individual trees. The default is "-svlb".

–C *folds*

When used alone, this option cancels the use of a seedfile (see the –V option). Instead training/test data set pairs are generated from the full data set in cross-validation style with *folds* number of folds. This is an inefficient way of doing cross validation on the full test set. If this option is followed by a –V or –v option, then samples are first selected (using the supplied seeds) and cross-validation is done with each of these samples (instead of the full data set), by sub-partitioning them in turn into a number of folds. This then returns a cross-validation estimate (with variances) of the statistics produced by running **ttest** without the –C option. (i.e. the report file produced with the –C option will be an estimate with variances of the statistics determined on the test sample without the –C option.)

–d Normally, statistics generated are appended to the existing report files. This option says to delete all report files at the very beginning so statistics collected represent those generated in just this run of **ttest**.

–D Echo the shell commands issued by **ttest**. This option is useful for debugging or learning how to use the system.

–k On abnormal exit, by default **mkclean** will be called, this option cancels this default.

–l *llopts*

Pass these options to **lstat** when generating the final summary report. The default is "-f 2 -v 1,2,3,7".

–O Output all results to stdio. With this option, the filename modifiers to the **–T** and **–R** options are assumed not to exist.

–R *ran trial prune name*

This is rather like running –T *trial prune name* five times and combining the output. Make 4 trees after the first tree using the "-R *ran*" option to **tgen**, as well as any options mentioned in *trial*. Only one –R option can be used, and it is incompatible with the –T option.

–T *trial prunelist namelist*

For each value *tprune* in the space delimited list *prunelist* and corresponding *name* from *namelist*, build a tree using the command "mktree -o *trial* -p *tprune*" and append the statistics gathered from tclass to the file "*stem*.trial.*size name*". **tclass** is ran with the –b option. Any "#" in *trial* or *tprune* for mktree will be replaced with a space character. The *namelist* argument is assumed not to exist if output is to stdio. The list of pruning options means that you can grow a tree once and then prune it in several different ways to test. When passng options "-n" to **tprune**, always use "#-n" because the **mktree** implementation causes a single "-n" to disappear. The –T option can occur multiple times (up to 6) and the trials will be run concurrently.

–V *seedfile*

One trial is ran for each train/test pair of the data set. A list of seeds are passed one at a time to **mkenc** to generate these different train test pairs. The *seedfile* is a file containing white space-separated integers to use as seeds. The default is "./seeds".

–v *seed-list*

Seeds are set from the space-separated list of integers supplied as argument.

SEE ALSO

mkbld(1), *tchar*(1), *tclass*(1), *tgen*(1), *mktree*(1),

NAME

xgraph – Draw a graph on an X11 Display

SYNOPSIS

xgraph [options] [=WxH+X+Y] [-display host:display.screen] [file ...]

DESCRIPTION

The *xgraph* program draws a graph on an X display given data read from either data files or from standard input if no files are specified. It can display up to 64 independent data sets using different colors and/or line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels, and a legend. There are options to control the appearance of most components of the graph.

The input format is similar to *graph(1G)* but differs slightly. The data consists of a number of *data sets*. Data sets are separated by a blank line. A new data set is also assumed at the start of each input file. A data set consists of an ordered list of points of the form "{directive} X Y". The directive is either "draw" or "move" and can be omitted. If the directive is "draw", a line will be drawn between the previous point and the current point (if a line graph is chosen). Specifying a "move" directive tells xgraph not to draw a line between the points. If the directive is omitted, "draw" is assumed for all points in a data set except the first point where "move" is assumed. The "move" directive is used most often to allow discontinuous data in a data set. The name of a data set can be specified by enclosing the name in double quotes on a line by itself in the body of the data set. The trailing double quote is optional. Overall graphing options for the graph can be specified in data files by writing lines of the form "<option>: <value>". The option names are the same as those used for specifying X resources (see below). The option and value must be separated by at bleast one space. An example input file with three data sets is shown below. Note that set three is not named, set two has discontinuous data, and the title of the graph is specified near the top of the file.

```
TitleText: Sample Data
0.5 7.8
1.0 6.2
"set one
1.5 8.9

"set two"
-3.4 1.4e-3
-2.0 1.9e-2
move -1.0 2.0e-2
-0.65 2.2e-4

2.2 12.8
2.4 -3.3
2.6 -32.2
2.8 -10.3
```

After *xgraph* has read the data, it will create a new window to graphically display the data. The interface used to specify the size and location of this window depends on the window manager currently in use. Refer to the reference manual of the window manager for details.

Once the window has been opened, all of the data sets will be displayed graphically (subject to the options explained below) with a legend in the upper right corner of the screen. To zoom in on a portion of the graph, depress a mouse button in the window and sweep out a region. *xgraph* will then open a new window looking at just that portion of the graph. *xgraph* also presents three control buttons in the upper left corner of each window: *Close*, *Hardcopy*, and *About*. Windows are closed by depressing a mouse button while the mouse cursor is inside the *Close* button. Typing EOF (control-D) in a window also closes that window. Depressing a mouse button while the mouse cursor is in the *Hardcopy* button causes a dialog to appear asking about hardcopy (printout) options. These options are described below:

Output Device
: Specifies the type of the output device (e.g. "HPGL", "Postscript", etc). An output device is chosen by depressing the mouse inside its name. The default values of other fields will change when you select a different output device.

Disposition
: Specifies whether the output should go directly to a device or to a file. Again, the default values of other fields will change when you select a different disposition.

File or Device Name
: If the disposition is "To Device", this field specifies the device name. A device name is the same as the name given for the -P command of lpr(1). If the disposition is "To File", this field specifies the name of the output file.

Maximum Dimension
: This specifies the maximum size of the plot on the hardcopy device in centimeters. *xgraph* takes in account the aspect ratio of the plot on the screen and will scale the plot so that the longer side of the plot is no more than the value of this parameter. If the device supports it, the plot may also be rotated on the page based on the value of the maximum dimension.

Include in Document
: If selected, this option causes *xgraph* to produce harcopy output that is suitable for inclusion in other larger documents. As an example, when this option is selected the Postscript output produced by xgraph will have a bounding box suitable for use with psfig.

Title Font Family
: This field specifies the name of a font to use when drawing the graph title. Suitable defaults are initially chosen for any given hardcopy device. The value of this field is hardware specific -- refer to the device reference manual for details.

Title Font Size
: This field specifies the desired size of the title fonts in points (1/72 of an inch). If the device supports scalable fonts, the font will be scaled to this size.

Axis Font Family and Axis Font Size
: These fields are like *Title Font Family* and *Title Font Size* except they specify values for the font *xgraph* uses to draw axis labels, and legend descriptions.

Control Buttons
: After specifing the parameters for the plot, the "Ok" button causes *xgraph* to produce a hardcopy. Pressing the "Cancel" button will abort the hardcopy operation. Depressing the *About* button causes Xgraph to display a window containing the version of the program and an electronic mailing address for the author for comments and suggestions.

xgraph accepts a large number of options most of which can be specified either on the command line, in the user's .Xdefaults or .Xresources file, or in the data files themselves. A list of these options is given below. The command line option is specified first with its X default or data file name (if any) in parenthesis afterward. The format of the option in the X defaults file is "program.option: value" where program is the program name (xgraph) and the option name is the one specified below. Option specifications in the data file are similar to the X defaults file specification except the program name is omitted.

=WxH+X+Y (Geometry)
: Specifies the initial size and location of the xgraph window. **-<digit> <name>** These options specify the data set name for the corresponding data set. The digit should be in the range '0' to '63'. This name will be used in the legend.

-bar (BarGraph)
: Specifies that vertical bars should be drawn from the data points to a base point which can be

specified with -brb. Usually, the -nl flag is used with this option. The point itself is located at the center of the bar.

–bb (BoundBox)
Draw a bounding box around the data region. This is very useful if you prefer to see tick marks rather than grid lines (see -tk).

–bd <color> (Border)
This specifies the border color of the *xgraph* window.

–bg <color> (Background)
Background color of the *xgraph* window.

–brb <base> (BarBase)
This specifies the base for a bar graph. By default, the base is zero.

–brw <width> (BarWidth)
This specifies the width of bars in a bar graph. The amount is specified in the user's units. By default, a bar one pixel wide is drawn.

–bw <size> (BorderSize)
Border width (in pixels) of the *xgraph* window.

–db (Debug)
Causes xgraph to run in synchronous mode and prints out the values of all known defaults.

–fg <color> (Foreground)
Foreground color. This color is used to draw all text and the normal grid lines in the window.

–gw (GridSize)
Width, in pixels, of normal grid lines.

–gs (GridStyle)
Line style pattern of normal grid lines.

–lf <fontname> (LabelFont)
Label font. All axis labels and grid labels are drawn using this font. A font name may be specified exactly (e.g. "9x15" or "-*-courier-bold-r-normal-*-140-*") or in an abbreviated form: <family>-<size>. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is "helvetica-12".

–lnx (LogX)
Specifies a logarithmic X axis. Grid labels represent powers of ten.

–lny (LogY)
Specifies a logarithmic Y axis. Grid labels represent powers of ten.

–lw width (LineWidth)
Specifies the width of the data lines in pixels. The default is zero.

–lx <xl,xh> (XLowLimit, XHighLimit)
This option limits the range of the X axis to the specified interval. This (along with -ly) can be used to "zoom in" on a particularly interesting portion of a larger graph.

–ly <yl,yh> (YLowLimit, YHighLimit)
This option limits the range of the Y axis to the specified interval.

–m (Markers)
Mark each data point with a distinctive marker. There are eight distinctive markers used by xgraph. These markers are assigned uniquely to each different line style on black and white machines and varies with each color on color machines.

–M (StyleMarkers)
Similar to -m but markers are assigned uniquely to each eight consecutive data sets (this corresponds to each different line style on color machines).

–nl (NoLines)
: Turn off drawing lines. When used with -m, -M, -p, or -P this can be used to produce scatter plots. When used with -bar, it can be used to produce standard bar graphs.

–p (PixelMarkers)
: Marks each data point with a small marker (pixel sized). This is usually used with the -nl option for scatter plots.

–P (LargePixels)
: Similar to -p but marks each pixel with a large dot.

–rv (ReverseVideo)
: Reverse video. On black and white displays, this will invert the foreground and background colors. The behaviour on color displays is undefined.

–t <string> (TitleText)
: Title of the plot. This string is centered at the top of the graph.

–tf <fontname> (TitleFont)
: Title font. This is the name of the font to use for the graph title. A font name may be specified exactly (e.g. "9x15" or "-*-courier-bold-r-normal-*-140-*") or in an abbreviated form: <family>-<size>. The family is the family name (like helvetica) and the size is the font size in points (like 12). The default for this parameter is "helvetica-18".

–tk (Ticks)
: This option causes *xgraph* to draw tick marks rather than full grid lines. The -bb option is also useful when viewing graphs with tick marks only.

–x <unitname> (XUnitText)
: This is the unit name for the X axis. Its default is "X".

–y <unitname> (YUnitText)
: This is the unit name for the Y axis. Its default is "Y".

–zg <color> (ZeroColor)
: This is the color used to draw the zero grid line.

–zw <width> (ZeroWidth)
: This is the width of the zero grid line in pixels.

Some options can only be specified in the X defaults file or in the data files. These options are described below:

<digit>.Color
: Specifies the color for a data set. Eight independent colors can be specified. Thus, the digit should be between '0' and '7'. If there are more than eight data sets, the colors will repeat but with a new line style (see below).

<digit>.Style
: Specifies the line style for a data set. A string of ones and zeros specifies the pattern used for the line style. Eight independent line styles can be specified. Thus, the digit should be between '0' and '7'. If there are more than eight data sets, these styles will be reused. On color workstations, one line style is used for each of eight colors. Thus, 64 unique data sets can be displayed.

Device The default output form presented in the hardcopy dialog (i.e. "Postscript", "HPGL", etc).

Disposition
: The default setting of whether output goes directly to a device or to a file. This must be one of the strings "To File" or "To Device".

FileOrDev
: The default file name or device string in the hardcopy dialog.

ZeroWidth
Width, in pixels, of the zero grid line.

ZeroStyle
Line style pattern of the zero grid line.

AUTHOR

David Harrison University of California

BUGS

- Zooming in on bar graphs doesn't work right.
- There is no way to produce hardcopy without running xgraph interactively.

Chapter 5

Installing IND

5.1 Introduction

IND is a suite of C programs and C shell scripts for building classifiers (i.e. , supervised learning). The code is provided (and sometimes even moderately documented) so you can develop your own extensions. IND was developed exclusively in a SUN workstation environment under various releases of SunOS UNIX, and can compile under "cc" or "gcc". The IND package really needs an X.11 interface or something similar to handle all the processing done by **mkbld**, **mktree**, and **ttest**. Note, ... First time users should see the companion note in "IND/Doc/Release.tex".

5.2 Overview of the IND Directory

Scripts: The "Scripts" directory contains all sorts of useful "csh" scripts which are usually documented in their beginning, and some have man entries.

Statlib: This contains the C library of statistical functions used in the various programs.

Treelib: This contains the C library of tree processing functions for read and write, grow, prune, etc.

Eglib: This contains the C library of example and contingency-table processing functions.

Util: This contains subdirectories with general system utilities such as sampling and encoding.

Trees: The tree programs are in this directory.

Man: The man entries for most things are included.

Doc: Various forms of documentation exist. The man entries are elsewhere. The subdirectory "course" details a 4 week 3rd year undergrad. course on trees, part of which is duplicated in the manual. Latex source for the IND manual is in "manual" which also contains a bibliography and the RIACS copyright. "Release.tex" is a LaTeX document that you should look at before anything else.

Include: The header and include files for the many data structures (trees, sets, examples, ...) are here.

Data: This is a sample data file directory which you can peruse to get an idea of data formatting, attribute file specifications, etc. Also, run the programs on these to test the system after installation.

5.3 Installing the Code

1. Check for machine compatibility by looking at

    - Lib/quickfit.c (top few #defines)
    - Include/Lib.h (last few lines)
    - Include/SET.h (at the top)

for potential storage type and alignment problems. The system has only been compiled on SUNs, so expect major problems on other UNIX machines. Some things like alarm() (in Trees/tgen.c) and ftime() and "struct timeb" (in random.c), and a few others are used, which tend to be UNIX version dependent.

2. Modify the primary "Makefile" to add your "BIN" to the file. Then run "make bin". This will modify all other IND Makefiles in the IND subdirectories so that they know where to place the bins.
3. Similarly, modify CC and CFLAGS in the primary "Makefile" and run "make cc" and "make cflags" respectively if you wish to change the factory-set options.
4. Compile using "make install". This will call make recursively in the various subdirectories to construct the ".o" files, the ".a" libs and then go on to make the programs and put them in your BIN.
5. This will also compile a slightly modified version of **xgraph** that requires certain "X11/include" files be on your system. If you won't be using this, then modify "IND/Makefile" so that **xgraph** would not be made, and never use the "x" option in interactive mode.
6. Add the "Scripts" directory to your own path.
7. Add the "Man" directory to your own MANPATH.
8. Run "rehash" since you've changed your path.
9. Try running some examples with "make test". Compare the output with "make.test.out".

5.4 Warnings

Software in the "IND/Bayes" directory (the simple Bayes classifier) has only been marginally tested. The full range of option combinations have not been tested and are not supported. In addition, options not listed in the man entries are not supported. For those looking at the code, bear in mind it is a research code and various features exist at different stages of development. The option trees and -J option to **tgen** doesn't have anytime search control so can be difficult to use.

Run-time trouble should really only be expected if you are using the "-J" option in **tgen** which builds option trees. This routine does a poorly controlled search so can consume large amounts of memory and time. Also the search may cut out prematurely, in which case the results are not indicative. See the "man" entry for **tgen** for details.

5.5 Planned Extensions

The code for IND is distributed free of charge (see the Copyright notice in Appendix B) for research purposes, to allow for all the tinkering researchers like to do on other people's algorithms. In this spirit, if you would like to make extensions for inclusion in future releases of IND, we would welcome discussions and suggestions. Bear in mind, your code will be distributed too, and maybe modified by others in future. Here is a list of wanted extensions that may well be under construction by the time you read this:

- X.11 interface to the package to reproduce the tasks of `ttest`, etc. , with a nice point and click interface, and to allow interactive tree learning [7].
- Extensions to the tree methods such as probabilistic approaches to multi-variate splits and missing values, incremental or large batch learning (e.g., [13]), etc.
- Other learning algorithms such as rule learning [30, 42], regression and/or back-propagation [11], learning Bayesian and/or Markov networks [8], and probabilistic variants of case-based or instance-based reasoning.
- Clean up the search control and interface to the Bayesian option trees (-J option in `tgen`), to make this powerful method more accessible. Smarter searching for option trees together with more compact summary tree to allow anytime search and to produce results more readily presentable.

5.6 Contact and Reporting Your Use of IND

Please notify us of your use of IND. We will then be able to inform you about enhancements, updates and bug fixes. We ask that you report any application you make of IND, describing the application and your analysis of the results. Please feel free to make suggestions about desirable improvements and extensions, and perceived problem areas. We regard such feedback as an essential element of the development process. For example, feedback on changes required to get the package running on other environments are welcome.

Contact details:

email:	ind@kronos.arc.nasa.gov
post:	IND Tree Package C/O Wray Buntine, RIACS and Code FIA Mail Stop 269-2 NASA Ames Research Center Moffett Field, CA, 94035 USA

Bibliography

[1] L.R. Bahl, P.F. Brown, P.V. de Sousa, and R.L. Mercer. A tree-based langauge model for natural language speech recognition. *IEEE Trans. on AS and SP*, 37(7):1001–1008, 1989.

[2] A.R. Barron and T.M. Cover. Minimum complexity density estimation. *IEEE Trans. on IT*, 37(4), 1991.

[3] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

[4] W.L. Buntine. Learning classification rules using Bayes. In *Proceedings of the Sixth International Machine Learning Workshop*, Cornell, New York, 1989. Morgan Kaufmann.

[5] W.L. Buntine. Learning classification trees. Technical Report FIA-90-12-19-01, RIACS and NASA Ames Research Center, Moffett Field, CA, 1990. Paper presented at Third International Workshop on Artificial Intelligence and Statistics.

[6] W.L. Buntine. Classifiers: A theoretical and empirical study. In *International Joint Conference on Artificial Intelligence*, Sydney, 1991. Morgan Kaufmann.

[7] W.L. Buntine. *A Theory of Learning Classification Rules*. PhD thesis, University of Technology, Sydney, 1991.

[8] W.L. Buntine. Theory refinement of Bayesian networks. In *Seventh Conference on Uncertainty in Artificial Intelligence*, Anaheim, CA, 1991.

[9] W.L. Buntine and T. Niblett. A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 1991. To appear.

[10] W.L. Buntine and D.A. Stirling. Interactive induction. In J. Hayes, D. Michie, and E. Tyugu, editors, *MI-12: Machine Intelligence 12, Machine Analysis and Synthesis of Knowledge*. Oxford University Press, Oxford, 1990.

[11] W.L. Buntine and A.S. Weigend. Bayesian back-propagation. *Complex Systems*. to appear.

[12] C. Carter and J. Catlett. Assessing credit card applications using machine learning. *IEEE Expert*, 2(3):71–79, 1987.

[13] J. Catlett. *Megainduction: machine learning on very large databases*. PhD thesis, University of Sydney, 1991.

[14] B. Cestnik, I. Kononenko, and I. Bratko. Assistant86: A knowledge-elicitation tool for sophisticated users. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning: Proceedings of EWSL-87*, pages 31–45, Bled, Yugoslavia, 1987. Sigma Press.

[15] P.A. Chou. *Applications of Information Theory to Pattern Recognition and the Design of Decision Trees and Trellises.* PhD thesis, Stanford University, 1988.

[16] P.A. Chou. Optimal partitioning for classification and regression trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1991.

[17] S.L. Crawford. Extensions to the CART algorithm. *International Journal of Man-Machine Studies*, 31(2):197–217, 1989.

[18] A. Hart. The role of induction in knowledge elicitation. *Expert Systems*, 2:24–28, 1985.

[19] S.L. Lauritzen and D.J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Roy. Statist. Soc. B*, 50(2):240–265, 1988.

[20] C.J. Matheus and L.A. Rendell. Constructive induction on decision trees. In *International Joint Conference on Artificial Intelligence*, pages 645–650, Detroit, 1989. Morgan Kaufmann.

[21] D. Michie. The superarticulacy phenomenon in the context of software manufacture. *Proc. Roy. Soc. (A)*, 405:185–212, 1986.

[22] D. Michie. Current developments in expert systems. In J.R. Quinlan, editor, *Applications of Expert Systems.* Addison Wesley, London, 1987.

[23] D. Michie. Statistical classifiers compared with decision-tree classifiers as applied to credit scoring. In J. Hayes, D. Michie, and E. Tyugu, editors, *MI-12: Machine Intelligence 12, Machine Analysis and Synthesis of Knowledge.* Oxford University Press, Oxford, 1990.

[24] J. Mingers. An empirical comparison of pruning methods for decision-tree induction. *Machine Learning*, 4(2):227–243, 1989.

[25] J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4):319–342, 1989.

[26] T. Niblett and I. Bratko. Learning decision rules in noisy domains. In M. A. Bramer, editor, *Research and Development in Expert Systems III*, pages 25–34. Cambridge University Press, 1987.

[27] G. Pagallo. Learning DNF by decision trees. In *International Joint Conference on Artificial Intelligence*, pages 639–644, Detroit, 1989. Morgan Kaufmann.

[28] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan and Kauffman, 1988.

[29] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[30] J.R. Quinlan. Generating production rules from decision trees. In *International Joint Conference on Artificial Intelligence*, pages 304–307, Milan, 1987.

[31] J.R. Quinlan. Simplifying decision trees. In B. Gaines and J. Boose, editors, *Knowledge Acquisition for Knowledge-Based Systems*, pages 239–252. Academic Press, London, 1988.

[32] J.R. Quinlan. Unknown attribute values in induction. In *Proceedings of the Sixth International Machine Learning Workshop*, Cornell, New York, 1989. Morgan Kaufmann.

[33] J.R. Quinlan, P.J. Compton, K.A. Horn, and L. Lazarus. Inductive knowledge acquisition: A case study. In J.R. Quinlan, editor, *Applications of Expert Systems*. Addison Wesley, London, 1987.

[34] J.R. Quinlan and R.L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80:227–248, 1989.

[35] J. Rissanen. *Stochastic Complexity in Statistical Enquiry*. World Scientific, 1989.

[36] J. Rissanen and Mati Wax. Algorithm for constructing tree structured classifiers, 1988. Patent Number 4,719,571.

[37] A. Shapiro. *Structured Induction in Expert Systems*. Addison Wesley, London, 1987.

[38] P. Smyth and R.M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Trans. on Knowledge and Data Engineering*, 1990.

[39] P. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[40] C.S. Wallace and P.R. Freeman. Estimation and inference by compact encoding. *J. Roy. Statist. Soc. B*, 49(3):240–265, 1987.

[41] C.S. Wallace and J.D. Patrick. Coding decision trees. Technical Report 151, Monash University, Melbourne, 1991.

[42] S.M. Weiss and N. Indurkhya. Reduced complexity rule induction. In *International Joint Conference on Artificial Intelligence*, Sydney, 1991. Morgan Kaufmann.

[43] S.M. Weiss and I. Kapouleas. An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In *International Joint Conference on Artificial Intelligence*, pages 781–787, Detroit, 1989. Morgan Kaufmann.

[44] J. Wirth and J. Catlett. Experiments on the costs and benefits of windowing in ID3. In *Fifth International Conference on Machine Learning*, pages 87–99, Ann Arbor, Michigan, 1988. Morgan Kaufmann.

Appendix A

Glossary

accuracy (of a classifier) A classifier takes a set of instances and classifies them. If it always classifies instances correctly then it is a perfect classifier. In domains where measurements and the classification itself may be noisy or uncertain and some key attributes may be missing, it is rarely possible to classify instances with 100% accuracy. The accuracy of a specific classify is long-run proportion of classifications it gets (or would get) correct. The error is the complement of the accuracy (error + accuracy = 1.0). This is sometimes estimated from a test set, but cannot by estimated from the training set. See "error estimates' in Section 3.3.4. The theoretical minimum error that can be achieved is termed the *Bayes error* and is the lowest possible long-run proportion of incorrect classifications achievable by any classifier.

attribute_file This text file contains the format description for the examples contained in the ".dta", ".bld", and ".tst" data files. The attribute_file can also specify utilities and constraints on how certain attributes may be tested in a tree. For example, it is possible to prevent attribute A from being tested unless attribute B has been tested as TRUE further up the tree. For more detail see the man page attributes(1) and look at the sample attribute_file "hypo.attr" in the directory /IND/Data/thyroid.

Bayes classifier A Bayes classifier, also called, "idiots Bayes", is a simple form of classifier that assumes the attributes are independent given the class. So to predict the boolean class c given boolean attributes a_1, a_2 and a_3, use the formula

$$Pr(c|a_1,a_2,a_3) = \frac{Pr(c)Pr(a_1|c)Pr(a_2|c)Pr(a_3|c)}{Pr(c)Pr(a_1|c)Pr(a_2|c)Pr(a_3|c)+Pr(\bar{c})Pr(a_1|\bar{c})Pr(a_2|\bar{c})Pr(a_3|\bar{c})} \ .$$

Using a logarithmic transformation, this becomes a linear model rather like a perceptron.

Bayesian averaging Using randomization methods, we can grow several different class probability trees, each of which should be quite good. Since we don't know which is the "best" tree, when classifying a new example, we can take the weighted average of the class probability vectors each tree assigns to the example. This means we are averaging over the individual recommendations of the trees. This and other Bayesian components of IND are explained in [7, 5].

Bayesian smoothing A single class probability tree can be pruned in many different ways. When classifying a single example, this means that the class probability vector assigned to the example can be taken from the leaf node or any of the interior nodes as well, depending on where pruning is done. Bayesian smoothing takes a different approach. Since we don't know the "best" place to prune the tree, Bayesian smoothing takes a weighted average of the class probability vectors that could be assigned along a given branch. The weights are determined using approximate Bayesian methods. This and other Bayesian components of IND are explained in [7, 5].

Bayes splitting rule This splitting rule is developed as a one-ply lookahead Bayesian estimate of the posterior probability of the split being "correct". It is similar to information gain when the sample size becomes large. This and other Bayesian components of IND are explained in [7, 5].

C4 C4 is the family of decision tree learning systems written by Ross Quinlan that superceded ID3 [31, 33]. Recent release C4.5 is sometimes available to the academic community.

CART stands for "Classification And Regression Trees" which is both a program and a book [3]. CART the program is a well known decision tree induction program with its roots in the statistics community. It was one of the first such programs available commercially and also one of the most successful. CART, the book, is an introduction to CART-style decision tree induction and a reference manual for running CART, the program. The first few chapters of the book are a reasonable introduction to some of the ideas in decision trees, such as handling missing attribute values and cros-validation. Through the appropriate choice of options, IND can be made to simulate CART-style decision tree induction.

cont (type description) This is an attribute type used to specify that an attribute represents a continuous variable, e.g. , a real-valued attribute on the interval [0,1]. See the man page for attributes(1) for more information.

class probability vector Probabilities for a set of mutually exclusive and exhaustive classes are represented as a vector of probabilities summing to 1.0. So for the classes *true* and *false* we might have the vectors (0.2,0.8), (0.64,0.36), etc. Class probability trees have these at their leaves.

context A context is an entry in an attribute file that restricts when an attribute may be tested. It is really a constraint on the structural form of trees that can be grown. For example, a context allows one to specify that one attribute may be tested only if another attribute is set TRUE. See the man page for attributes(1) for more detail.

cost-complexity pruning A way of trading off the size of a decision tree (its "complexity") against the accuracy of the decision tree (its "cost"). More formally, cost-complexity pruning seeks to minimize $SE + \alpha L$, the sum of the substitution error estimate SE (the number of errors the tree makes when tested on the training set) with a constant, α, multiplied by the number leaves in the tree L. If $\alpha = 0$, then there is no penalty for a large tree. As α gets larger, the penalty for larger trees increases. For each fixed value of α, there is an optimal pruned subtree of the original tree that minimizes this sum. Thus, by varying α we can generate a nested sequence of (pruned) trees, each of which is smaller—and potentially less accurate—than the tree preceding it in the sequence. A test set may then be used to estimate the prediction error of each tree, and the tree with the lowest prediction error is then selected from the sequence. To summarise, cost-complexity pruning allows an ordered sequence of pruned subtrees to be created, each of which represents a somewhat different tradeoff of complexity vs. accuracy. A test set or cross validation is then used to pick the subtree that yields the best prediction accuracy. See chapter 3.3.4 for more detail.

cross-validation A way of estimating the accuracy of an induction method (in this case a tree induction program). This is done by repeatedly holding out a small subset of the available data, training on the remainder, and then testing the result of induction (in this case the decision tree) by running it on the held out test set. The estimate is the average of the accuracies on the held out test sets. This is a good ("unbiased"), though computationally expensive, means of estimating predictive accuracy. K-fold cross validation does this by splitting the data set into K pieces, and then using $K - 1$ of them for training and the remainder for testing, to yield K different train-test pairs. See chapter 3.3.4 for more detail.

cut point When an ordered attribute (e.g. , an integer or real valued attribute) is used at a node test, the value at which to split the examples is the cut point for that test. For example, in a decision tree that dealt with fever, if some interior node has the test "temperature > 100", then "100" is the cut point for the test on the real-valued attribute "temperature".

decision node A decision tree contains two kinds of nodes: test nodes and leaf nodes. Decision nodes occur only at the leaves of the tree and represent the class to be assigned to any example that reaches that node. Thus, if an example reaches a leaf node labeled with the class "has_fever", then that example is classified as belonging to the class "has_fever".

lookahead Decision trees are typically *grown* using greedy search: at every node to be expanded by introducing an attribute test, greedy search considers how beneficial each test appears to be if we grew the tree one level more with that test. In effect, the algorithm is doing local hill-climbing where every decision about what to do next depends only upon examining the nearest possibilities.

Greedy node expansion works reasonably well in practice, and makes tree induction efficient (because very few options have to be considered at any one time), but may not lead to optimal trees. Sometimes the best attribute test to install at a node is one that is not best in the short term, but one that would be better in the *long term*. Lookahead considers trees of some bounded depth, say 2 or 3 deep, that are likely to be candidates to grow from the node currently being expanded. It evaluates the expected performance of these trees, and picks the best tree. It then installs just the first attribute test from the root of this best tree as the attribute test for the node being expanded.

Depth-bounded lookahead is akin to lookahead in game playing programs (e.g., games like chess). Instead of just picking a move based on an examination of the current board, most chess playing programs lookahead several ply to examine the consequences of each possible move, and to better evaluate which move is best to make now. Depth-bounded lookahead can increase the performance of the resulting tree. But there is a computational cost to be paid for this advantage: depth-bounded lookahead must examine plausible trees of some fixed size for each node it expands. This is certainly more expensive than just "looking ahead" 1 node as with standard tree induction, and becomes prohibitively expensive as the depth of lookahead becomes larger than 3. The implementation of IND uses a beam search when looking ahead.

expected accuracy (of a classifier) A classifier takes a set of instances and classifies them. If it always classifies instances correctly then it is a perfect classifier. In real domains where measurements and the classification itself may be noisy, it is rarely possible to classify instances with 100% accuracy. The *expected accuracy* of a classifier is the expected percentage of future instances that the classifier will classify correctly. Note that the expected accuracy is not a measure of how well the classifier classified the examples it was trained on, as this would typically significantly overestimate the classifier's performance on new data. Expected accuracy is sometimes estimated by testing the classifier on a test set of data intentionally held out of the training set. Bayesian methods use a more complex formula involving the predictive distribution of unseen examples to estimate expected accuracy.

GINI index of diversity A candidate attribute test is evaluated by measuring how well it separates the examples at that node into branches that consist of relatively pure classes. For example, an ideal attribute test (for the two class case) is one that sends all members of one class down one branch and all members of the other class down the other branch. Attribute tests are rarely ideal, so some measure of how well the test separates the classes is needed to evaluate how good the test is. One such measure is the *GINI index of diversity*. See Section 3.3.3.

information gain A candidate attribute test is evaluated by measuring how well it separates the examples at that node into branches that consist of relatively pure classes. For example, an ideal attribute test (for the two class case) is one that sends all members of one class down one branch and all members of the other class down the other branch. Attribute tests are rarely ideal, so some measure of how well the test separates the classes is needed to evaluate how good the test is. One such measure is the *Information gain* popularized by Quinlan [29]. See Section 3.3.3.

leaf node A tree contains interior nodes and *leaf* nodes. In a decision tree the leaf nodes represents the classification returned by the decision tree. For non-Bayes decision trees, a leaf node typically represents a single class, and any instance that ends up in that leaf node is assigned that class. In Bayes decision trees, each leaf node represents the assignment of a probability that the instance belongs to each possible class. For example, in a Bayes tree some leaf node might repreent the assignment that the instance is in class HAS_FEVER with probability 0.99 and is in class NO_FEVER with probability 0.01.

logical data set Some data sets represent situations where all possible combinations of feature values along with the correct classification can be enumerated. Typically these situations arise with data sets derived from certain "logic" functions such as learning a ten bit parity function. IND treats exhaustively enumerated data sets differently than non-exhaustively enumerated sets (see *mkbld*). Typically, the goal with enumerated data sets is to see if the induction algorithm can learn the already known concept (or perhaps how efficiently it learns the concept). Moreover, most logical data sets are brittle—missing a few examples usually causes a different concept to be induced. For these reasons, IND does not break logical data sets into sampled training and test sets. Instead, it uses the entire data set for both the training and test set. A separate file extent, ".all" is used by IND to indicate that a set of examples is exhaustive and should not be partitioned.

MDL/MML The minimum description length principle, and the related minimum message length principle. These principles use "encoding length" to measure the quality of hypotheses. An "encoding length" for a tree learned from a sample consists of a code for the tree together with a code for the classifications in the sample constructed on the basis of knowing the tree and the example types. These principles are often considered as approximate Bayesian methods since a non-redundant code length is the logarithm of some probability measure. See [40, 2].

mean square error (of a classifier) The "true" mean square error for a class probability tree is the average of the squared distances between the "true" class probability vector for an example and the class probability vector assigned to the example by the tree. This is approximated and reported by **tclass** as the half-Brier score, which is evaluated on the test set as

$$\sum_{\textit{tree correct on example } i} (\theta(i) - 1.0)^2 + \sum_{\textit{tree incorrect on example } i} \theta(i)^2 \quad ,$$

where $\theta(i)$ is the class probability the tree assigns to the i-th example.

minimum errors subtree The goal of pruning is to find the subtree of the induced decision tree that is expected to perform best on future examples. The pruned subtree that yields the fewest errors on a test set (i.e. , a set not used for training the tree) is the *minimum errors subtree* and is usually what we want the induction program to return.

misclassification matrix When a classifier classifies a set of examples, some of the examples will probably be misclassified. The misclassification matrix is a table that lists the correct classes along one axis and the classification derived from the classifier on the other axis. Each entry i, j in the table is the number of examples of true class i that were classified as class j. If the classifier is perfect, then only the diagonal entries are nonzero. The misclassification matrix is useful because it provides more information than the error rate alone; the matrix tells what kinds of errors are being made.

partition (a data set) Typically a user of an induction program has a single, hopefully large, set of examples from the domain. Usually, it is difficult to acquire additional examples, so the user has to make do with the set in hand. But the need to test the decision tree on a set of examples on which it was not trained (in order to accurately estimate the predicted performance) means that the original set of examples must be *partitioned* into a training and a test set. So partitioning is a way of splitting one large set of examples into two or more smaller sets that will be used for training and testing.

pessimistic pruning A way of pruning a decision tree. The basic approach is to grow the tree to full size. Then, for each test node, compute the resubstitution error estimate (the error of the tree rooted at that node as measured on the original training data) and the standard error of this estimate. Prune this tree (i.e. , replace it with a leaf node) if the confidence interval for the resubstitution error (equal to the resubstitution error plus some number of standard errors, typically 1) includes the expected resubstitution error of the node as a leaf node. The intuition behind the technique is to prune away subtrees that do not perform significantly better than a leaf node would at that position in the tree. See section 3.3.4. This pruning method was used in early versions of C4 and is implemented in the IND package.

posterior (of a decision tree) Posterior of a true is a measure of the quality of the true given in units of probability. Posteriors in IND are reported in log-probabilities. The IND system believes that a tree with log-posterior -75.4 is approximately $e^{2.1}$ times more likely to be the "true" tree than a tree with log-posterior -77.5. By comparison, a tree with log-posterior -175.4 can be safely ignored. Trees with similar relatively log-posteriors are alternative candidates. In tree averaging, done using the -J option in **tgen**, trees with high log-posteriors are collected and stored in an and-or structure.

pruning A tree grown on a training set can "overfit" that training set. That is, some of the branches in the tree that are useful for discriminating examples in the training set may not work well on unseen examples. In effect, the tree has achieved increased performance on the training data by making distinctions that may not be warranted in the domain itself. (Keep in mind that if the set of training examples is consistent we could always build a decision tree that classified the training examples perfectly by making each training example end up in its own leaf node which would then be assigned that examples class. But this "perfect" tree might perform quite poorly on the new examples it had not been trained on.)

Pruning is a process of eliminating many of the unwarranted subtrees lower in the tree by more carefully examining the effect of all subtrees on the estimated performance of the decision tree on unseen examples. Pruning is done after the full tree has been grown instead of while growing the tree because it would be difficult to evaluate the usefulness of some new test at a node without also knowing the tests that would be in the tree under it. That is, pruning is most accurate when the full subtree rooted at each test node can be evaluated. See section 3.3.4 for detail about different approaches to pruning.

test set A tree grown on a training set typically performs better on that training set (i.e. , makes fewer errors) than it will perform on future instances for which it was not trained. This is the result of overfitting the training set and is difficult to fully prevent. Because of this, the accuracy of the decision tree on the training data is optimistic and not indicative of the performance one is likely to achieve with the tree when applying it to future instances. Since we typically wish to evaluate the likely performance of the tree before actually using it to make real decisions, it is common to partition the data available into a training set and a test. The tree, then, is induced on the training set and subsequently tested on the test set. Since the test set was not used when the tree was induced, evaluating the decision tree on the test set provides an unbiased estimate of the tree's expected performance on new instances from the domain. Of course, there are other ways of evaluating the quality of a tree that don't require keeping aside a test set: cross validation, Bayesian methods and MDL/MML. These usually make more efficient use of available data, so give better results on smaller samples.

training set See **test set**.

utilities specification (in attribute_file) Utilities (see below) for the domain can be described in the attribute_file. This allows **tclass** to more appropriately choose the best class. Based on the predicted class probabilities, IND seeks to maximise expected utility.

utility Not all mistakes cost the same. In medical diagnosis, the cost of false positives (predicting someone has a disease when they don't) may be the cost of a few drugs but the cost of true negatives (predicting someone doesn't have a disease when they do) may mean death or permanent damage. These costs are termed utilities in decision theory (in fact, cost is the negative of utility) and ones seeks to make a prediction to maximize expected utility.

WRAY An acronym for Wray's Recursive Arbor Yielder, an alternate name for IND.

WRAY'S An acronym for Wray's Recursive Arbor Yield'n System (or Software), yet another name for IND.

Appendix B
Copyright

THE RIACS SOFTWARE POLICY

January 1988

1. INTENT

This section is only a summary of the intent of this document, and does not represent the actual software distribution policy of the Research Institute for Advanced Computer Science (RIACS).

- The software written at RIACS comes with absolutely no warranty. RIACS distributes research and prototype, but no production, software.
- The software written at RIACS will contain one of two copyright notices, indicating whether or not it may be redistributed. Prototype software will contain the "restricted distribution" copyright, and is for testing and comments only. Research software will contain the "reserved distribution" copyright, and may be given to other parties.
- Any software written at RIACS may be modified and duplicated, however if you modify any file you must clearly state in the file when it was altered, and who altered it.
- You are not allowed to charge for the licensing of any RIACS software you may redistribute, nor are you allowed to charge more than a nominal fee for making the redistribution.

2. THE RESERVED COPYRIGHT

Everyone is granted permission to copy, modify, and redistribute any RIACS software containing the following RIACS copyright notice, hereinafter referred to as the Reserved RIACS Copyright, but only under the RESERVED conditions stated in sections 2.1, 2.2, and 2.3.

> Copyright © 1987 Research Institute for Advanced Computer Science. All rights reserved. The RIACS Software Policy contains specific terms and conditions on the use of this software, and must be distributed with any copies. This file may be redistributed. This copyright and notice must be preserved in all copies made of this file.

2.1. Reserved Duplication

You may duplicate any source code containing the Reserved RIACS Copyright as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice such as "Copyright © 1987 RIACS," containing the year of last change and name of copyright holder for the file in question; and keep intact the notices on all files that refer to this Software Policy.

You may duplicate any software containing the Reserved RIACS copyright or any portion of it in compiled, executable or object code form.

2.2. Reserved Modification

You may modify your copy or copies of source code containing the Reserved RIACS Copyright provided that you cause the modified files to carry prominent notices stating who last changed such files and the date of any change.

2.3. Reserved Distribution

The whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of software, or any part thereof, containing the Reserved RIACS Copyright, must be made available to all third parties on terms identical to those contained in this Software Policy.

You may charge a distribution fee for the physical act of transferring such software, and you may at your option offer warranty protection, which is not mandatory, in exchange for a fee. You may not charge a fee for the licensing of reserved software.

You may distribute any software containing the Reserved RIACS copyright or any portion of it in compiled, executable or object code form, provided that you cause each such copy of this software to be accompanied by a copy of this Software Policy document; and in addition do the following:

- cause each such copy of this software to be accompanied by the corresponding machine-readable source code; or
- cause each such copy of this software to be accompanied by a written offer, which is good for at least one year, to give any third party free (except for a nominal shipping charge) machine readable copy of the corresponding source code; or
- in the case where you are a recipient such software in compiled, executable or object code form (without the corresponding source code) you shall cause copies you distribute to be accompanied by a copy of the written offer for source code which you received along with your copy such software.

3. THE RESTRICTED COPYRIGHT

Everyone is granted permission to copy and modify, but not to redistribute, any RIACS software containing the following RIACS copyright notice, hereinafter referred to as the the Restricted RIACS Copyright, additionally subject to the RESTRICTED conditions stated in sections 3.1, 3.2, and 3.3.

> Copyright © 1987 Research Institute for Advanced Computer Science. All rights reserved. The RIACS Software Policy contains specific terms and conditions on the use of this software. In particular, this software may not be distributed to any other party without explicit permission from RIACS.

3.1. Restricted Duplication

You may duplicate any source code containing the Restricted RIACS Copyright as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice such as "Copyright © 1987 RIACS," containing the year of last change and name of copyright holder for the file in question; and keep intact the notices on all files that refer to this Software Policy.

You may duplicate any software containing the Restricted RIACS copyright or any portion of it in compiled, executable or object code form.

3.2. Restricted Modification

You may modify your copy or copies of source code containing the Restricted RIACS Copyright provided that you cause the modified files to carry prominent notices stating who last changed such files and the date of any change.

3.3. Restricted Distribution

The whole of any work that in whole or in part contains or is a derivative of software, or any part thereof, containing the Restricted RIACS Copyright, may not be made available to any other party, in any form or medium, with the exception that all such software will be made available to RIACS.

4. NO WARRANTY

This software is distributed without any WARRANTY from the National Aeronautics and Space Administration (NASA), the University Space Research Association (USRA), RIACS, or any person associated with these organizations. These parties DO NOT accept responsibility for the consequences of anyone using any of this software, for whether it serves any purpose, or for its working order.

Because all software distributed by RIACS is either research or prototype software, and is free of charge, NASA, USRA, RIACS, AND ANY PERSON ASSOCIATED WITH THESE ORGANIZATIONS PROVIDE ABSOLUTELY NO WARRANTY TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, ALL SUCH SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE. The entire risk as to the quality and performance of all such software is with you. Should any of this software prove defective, you assume the cost of all necessary servicing, repair, or correction.

NASA, USRA, RIACS, or any other party who may modify or redistribute software received from RIACS as permitted above shall in no event be liable for any claims or demands by you or any other party, or any other claim or demand against NASA, USRA, RIACS, or other party due to or arising out of your use or inability to use any such software, and you agree to indemnify and hold NASA, USRA, RIACS, and any other party who may modify or redistribute software received from RIACS as permitted above harmless against all such claims.

5. TERMS

By accepting software and this Software Policy document from RIACS, in any form or medium, you are accepting the terms and conditions set forth in this document.

You may not duplicate, license, distribute, or transfer any software containing a RIACS copyright except as expressly provided under this Software Policy. Any attempt to otherwise duplicate, license, distribute, or transfer this software will terminate your rights under this agreement. However, parties who have received software from you with this Software Policy document will not have their rights terminated so long as such parties remain in full compliance with the terms and conditions herein.